2013

Accelerating Computational Algorithms

Michael Risley
Virginia Commonwealth University

Accelerating Computational Algorithms

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science at Virginia Commonwealth University.

by

Michael Risley
Master of Science

Director: Angela M. Reynolds, Assistant Professor
Department of Mathematics and Applied Mathematics

Virginia Commonwealth University
Richmond, Virginia
December 2013

Acknowledgment

William Curry, Carl Beckelheimer, and Joseph Gibson for continuous professional support, Dr. Dewey Taylor for copious encouragement, and Drs. Angela Reynolds and Ghidewon for making mathematical programming fun.

Contents

Abstract

ACCELERATING COMPUTATIONAL ALGORITHMS

By Michael Risley, Master of Science.

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science at Virginia Commonwealth University.

Virginia Commonwealth University, 2014.

Director: Angela M. Reynolds, Assistant Professor, Department of Mathematics and Applied Mathematics.

Mathematicians and computational scientists are often limited in their ability to model complex phenomena by the time it takes to run simulations. This thesis will inform interested researchers on how the development of highly parallel computer graphics hardware and the compiler frameworks to exploit it are expanding the range of algorithms that can be explored on affordable commodity hardware. We will discuss the complexities that have prevented researchers from exploiting advanced hardware as well as the obstacles that remain for the non-computer scientist.

Introdution

In this thesis we will introduce the OpenCL extension to the C computing language currently under development by multiple computer hardware designers including leading central processing unit (CPU) and graphics processing unit (GPU) manufacturers. We will discuss an abreviated history of computer development as it pertains to scientific computing and how this has led to utilization of the GPU for general purpose computing. We will discuss the differences between sequential CPU design, parallel GPU design, and how the two are starting to converge. Then we will briefly discuss single vs. double precision floating point arithmetic. In Chapter 2, we will examine a simple example of a parallel algorithm and its implementation in three languages. We will test all three implementations on two machines. The first is a small workstation with the CPU and GPU integrated into a single unit that chip manufacturer Advanced Micro Devices (AMD) calls an Accelerated Processing Unit (APU). The second is a larger workstation with an Intel-i7 series CPU and an Nvidia GPU located on a discrete graphics card with it's own dedicated high speed memory. The OpenCL related specifications for the two machines are in Appendix C. In Chapter 3 we will look at a common image manipulation example that can be executed on both machines. In Chapter 4 we will look at a more computationally intensive example that will be executed on the more powerful machine to allow for double prescision floating point arithmetic.

1.1 History of Parallel Architectures

To motivate why researchers should understand about parallel computing architectures, it helps to understand why they became necessary. Scientific computing was originally done on specialized hardware. Unix workstations typically running MIPS or Sparc processors could run simulation or computer aided design programs but they were expensive machines. Running large scale computationally intensive applications required access to "super-computer" class hardware. These were custom designed machines that were out of reach for most univeristies and completely unimaginable for individuals to own.

Over the ensuing decades, hardware got significantly faster and networking equipment became inexpensive. This allowed the construction of Beowulf clusters. Beowulf clusters are small collections of "commodity off the shelf" (COTS) hardware, networked together into a single system image, or computer. This was still too expensive for individuals but within reach of research departments. The following example illustrates the increase in computational power realized and how far desktop hardware has come. In a book on state of the art acceleration of computational models [6], a researcher recounts an example of moving a simulation that took 200 hours to complete on a high end Unix workstation to a cluster of 32 commodity linux boxes reducing the run time to 97 minutes. The cluster had 32 single core processors operating at 900 Mhz with 512MB of ram each. Although this example is from 2003, this is the same amount of ram in a 2011 desktop machine which has 4 CPU cores operating at 2.7 Ghz which will be used as one of the test machines in this thesis. This is not a particularly powerful machine by today's standards. It was assembled for well under 1,000USD with components selected primarily for low power consumption and low noise operation. For an amount of money that would be well within a lot of departmental budgets, researchers now have access to more parallel computational

power than could be had for 10 times the expense in 2005. The development of parallel CPU architectures, massively parallel graphics processors, unprecedented multi-level memory availability, and compilers that can utilize these resources, have lead to what recently was considered super-computer level resources being available to individual researchers.

To understand why every one is not running highly parallel code on massively Beowulf clusters requires understanding the obstacles to utilizing networked systems. Clusters require specialized computer programming skills and careful algorithm design. To exploit networked resources the programmer must take care that the time processors spend communicating results to each other does not exceed the time spent performing computations [14]. Although certain problems (especially those involving sparse matrices) lend themselves to cluster computing, the detailed knowledge of computer memory management has kept cluster programming in the domain of experts. The programming framework discussed in this thesis is accessable to researchers with basic programming skills.

The trend toward multicore processors has increased the need for parallel computing. As technonlogy advanced, smaller and smaller circuits became feasible and computers became faster simply by increase the frequency at which the processor operates. This has run into a natural limit at very small ciruit sizes (Intel is currently fabricating CPUs with features only 22 nanometers in width). The decrease in feature size has brought us to the point that increasing the CPU clock rate would require a reduction in voltage to a level that leads to increased error rates [5]. As it became clear that frequency could not be increased indefinitely, manufacturers turned to multicore processors.

Today's most powerful computers are basically large scale Beowulf designs with custom low-latencey networks between processors with each processor containing multiple CPU

cores. In 2005 the median number of CPU cores needed to make the top 500 supercomputer list was 768, in 2013 it was over 17,000 [1]. In 1995 all of the computers on the top 500 list had single core processors, in 2013 over 50% had 8 CPU cores per processor. And the most power efficient supercomputer, TITAN at Oak Ridge National Labs, contains over 18,000 NVIDIA GPU compute units in addition to almost 300,000 CPU cores. From their website:

> The combination of CPUs and GPUs will allow Titan and future systems to overcome power and space limitations inherent in previous generations of high-performance computers. Because they handle hundreds of calculations simultaneously, GPUs can go through many more than CPUs in a given time. Yet they draw only modestly more electricity. By relying on its 299,008 CPU cores to guide simulations and allowing its Tesla K20 GPUs, which are based on NVIDIA's next-generation Kepler architecture to do the heavy lifting, Titan will be approximately ten times more powerful than its predecessor, Jaguar, while occupying the same space and drawing essentially the same level of power. When complete, Titan will have a theoretical peak performance of more than 20 petaflops, or more than 20,000 trillion calculations per second. This will enable researchers across the scientific arena, from materials to climate change to astrophysics, to acquire unparalleled accuracy in their simulations and achieve research breakthroughs more rapidly than ever before [3].

1.2 Processor Architectures

Graphics processing in which the same operation is performed on thousands or even millions of individual data points is an inherently parallelizable process. Increasingly demanding graphical user environments, higher resolution displays, and complex computer games have led to mass market production of very fast parallel processors. The availability of

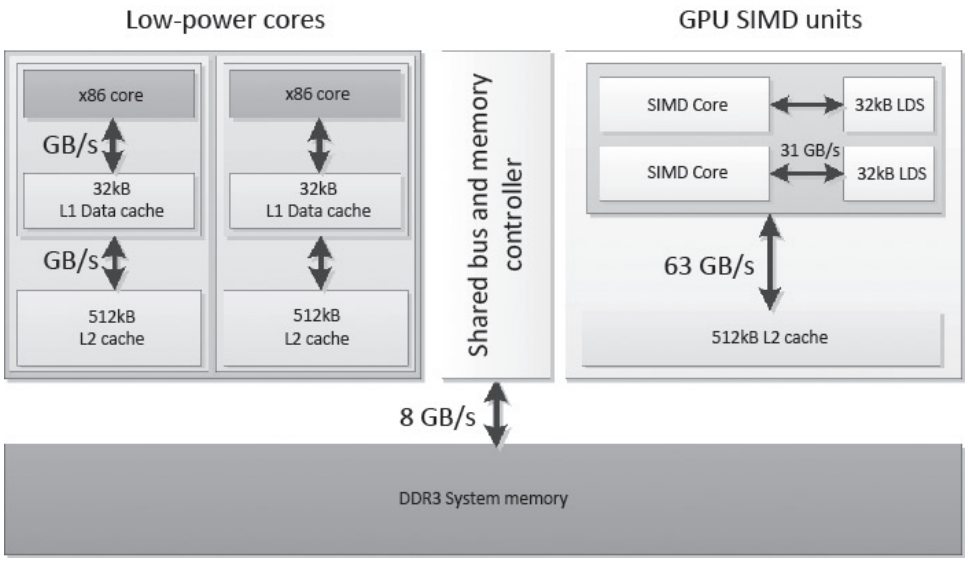

Figure 1.1: AMD APU processor similar to test machine 1 [5]

massively parallel hardware has in turn led to attempts to utilize the GPU as processing units. Manufacturers began to support this with propritery frameworks like CUDA from NVIDIA and AMD's ACML framework. Customer demand for a single programming interface has led to Khronos group being formed to standardize a C language extension called OpenCL to allow execution on GPU hardware and precise memory management. We will use this language to illustrate the level of pragramming skill required to utilize GPU computing capabilities and the speed increases that can be realized. A simple vector addition program is shown to familiarize the reader with the syntax involved and to show the different amount of work involved using OpenCl in C as opposed to the available C++ or Python PyOpenCL frameworks.

We will then demonstate an image rotation example ported to graphics hardware on an integrated system with an AMD APU processor see Figure 1.1, and Singular Value Decomposition on an Intel based system with a discrete NVIDIA GPU, see Figure 1.2.

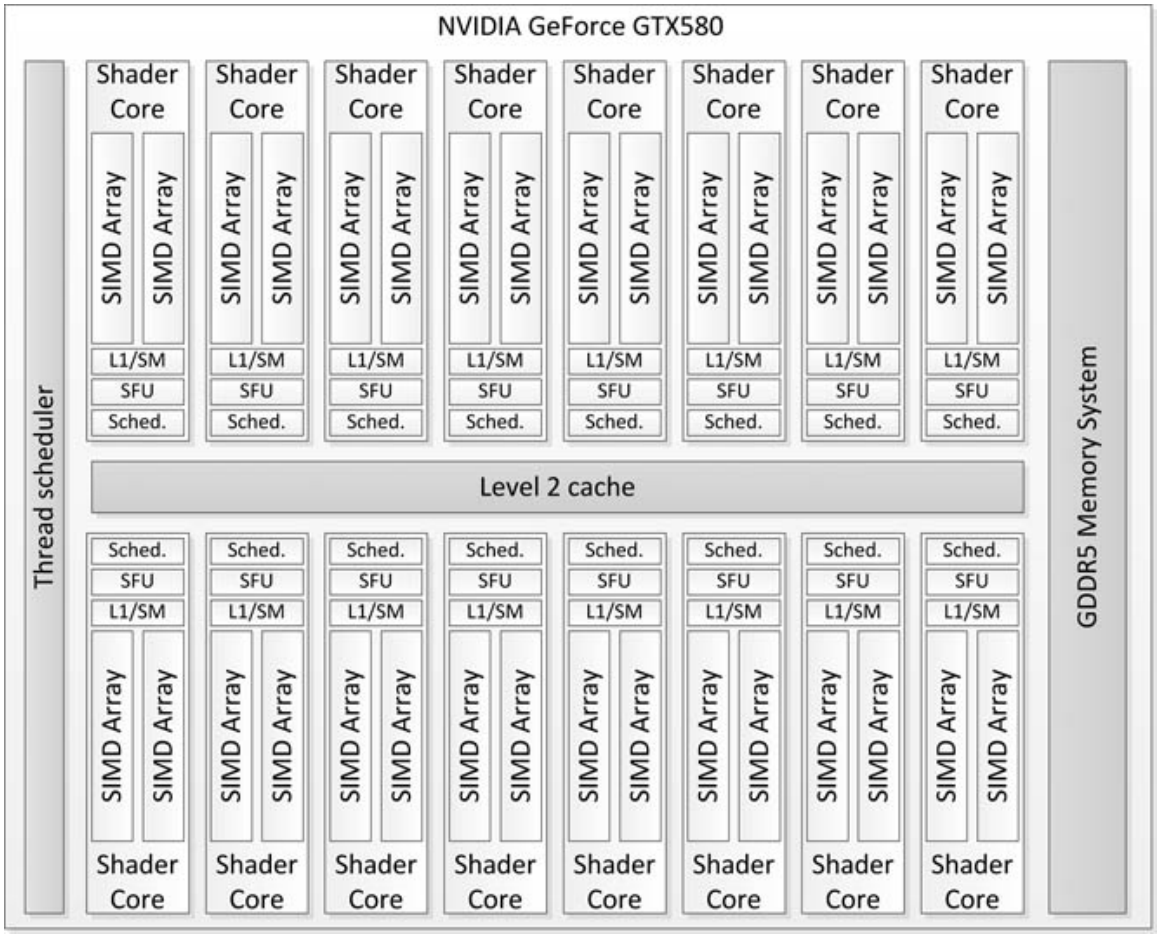

Figure 1.2: Nvidia GPU used in test machine 2 [5]

1.3 Floating Point Precision

The question of accuracy is sometimes raised in the context of GPU computing since they have traditionally supported only singel precision floating point arithmatic unlike CPUs which have double precision support. To understand the discussion requires some background on machine arithmetic. Physical machines are capable of storing only a finite number of digits to represent an approximation of a real number. For any real number with a binary representation longer than the physical storage limit (e.g. any irrational number), rounding becomes necessary.

IEEE standard 754 [15] defines how real numbers are stored and rounded in compliant computer architectures. IEEE 754 defines two floating point (real numbers as opposed to integers) data types for C programming, the float and double. Floats are represented by 32 bits of precision and double by 64. These are commonly referred to as single and double precision respectively. The problem of accuracy arises when two or more numbers have to be used in a computation. Consider a simple equation:

$$a+b+c=d$$

To calculate d, we must first add a to b or b to c. When working with real numbers these results are always the same, i.e.

$$(a+b)+c=a+(b+c).$$

But if we are rounding each number we are actually performing,

$$\mathrm{rnd}(a)+\mathrm{rnd}(b)+\mathrm{rnd}(c).$$

So, order of operations becomes important because the computer must also round intermediate results and

$$\text{rnd}(\text{rnd}(a)+\text{rnd}(b))+\text{rnd}(c) = \text{rnd}(a)+\text{rnd}(\text{rnd}(b)+\text{rnd}(c))$$

is not necessarily true. Obviously, the more precision we use to store real numbers, the less likely rounding differences are to matter. Early graphics hardware did not support double precision arithmetic because it was unnecessary in graphics processing. As graphics have become more complicated and GPUs are being used for scientific computing, manufacturers have added double precision support. The test machines used in this discussion contain an embedded GPU in the AMD APU that does not support double precision. The NVIDIA graphics card in the other machine does support double precision arithmetic and is used exclusively for the matrix factorization example.

Parallel Vector Addition

2.1 "hello world"

Vector addition (vadd) is considered the "hello world" problem of OpenCL. This is the simplest example of a function that is traditionally programmed in a sequential algorithm with an inner control loop and an outer control loop. Consider the case of adding to vectors a and b of real numbers of length n, with vector addition defined in the usual way:

$$\langle a_1, a_2, \ldots, a_n \rangle + \langle b_1, b_2, \ldots . b_n \rangle = \langle a_1 + b_1, a_2 + b_2, \ldots, a_n + b_n \rangle$$

In a sequential language, a programmer would define three vector objects, a, b, c and an index variable $i = 1$ and then while $i < n$ compute,

$$c_i = a_i + b_i$$

It should be obvious that we can execute as many additions as hardware allows simultaneously.

2.2 Vadd in C

The following code for this vector addition example is from Hands on OpenCL tutorial [10]. The C code is provided to demonstrate what is necessary to program even a simple example directly in C with the OpenCL extensions and to illustrate why researchers have been

reluctant to embrace GPU computing prior to the development of higher level frameworks. To researchers who normally program in higher level languages, C can be quite verbose and intimidating but it allows explicit control of memory utilization. In order run this program, the OpenCL language extension need to be installed along with the usual C and C++ compiler tools. As of 2013, Apple supports the Intel OpenCL implementation on Macintosh computers that have Intel CPUs with integrated Intel HD graphics. The necessary libraries should be installed if the developer tools package is installed. If the researcher is running Linux or Windows, NVIDIA and AMD provide OpenCL enabled video drivers and vendor specific OpenCL libraries. The manufacturers installation instructions should be followed for Windows machines. Linux users should follow their distribution's package managenment instructions. After the necessary libraries are installed, the source code should be compiled in the usual manner but it may be necessary to specify the location of the OpenCL library.

vaddCode/vadd_c.c

```
//------------------------------------------------------------------------------
//
// Name:        vadd.c
//
// Purpose:     Elementwise addition of two vectors (c = a + b)
//
// HISTORY:     Written by Tim Mattson, December 2009
//              Updated by Tom Deakin and Simon McIntosh-Smith, October 2012
//              Updated by Tom Deakin, July 2013
//
//------------------------------------------------------------------------------


#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#ifdef __APPLE__
#include <OpenCL/opencl.h>
#include <unistd.h>
#else
#include <CL/cl.h>
#endif

//pick up device type from compiler command line or from
//the default type
#ifndef DEVICE
#define DEVICE CL_DEVICE_TYPE_DEFAULT
#endif
```

```
extern double wtime();       // returns time since some fixed past point (wtime.c)
extern int output_device_info(cl_device_id );
char* err_code (cl_int);

//------------------------------------------------------------------------------

#define TOL    (0.001)   // tolerance used in floating point comparisons
#define LENGTH (1024)    // length of vectors a, b, and c

//------------------------------------------------------------------------------
//
// kernel:  vadd
//
// Purpose: Compute the elementwise sum c = a+b
//
// input: a and b float vectors of length count
//
// output: c float vector of length count holding the sum a + b
//

const char *KernelSource = "\n" \
"__kernel void vadd(                                                 \n" \
"   __global float* a,                                               \n" \
"   __global float* b,                                               \n" \
"   __global float* c,                                               \n" \
"   const unsigned int count)                                        \n" \
"{                                                                   \n" \
"   int i = get_global_id(0);                                        \n" \
"   if(i < count)                                                    \n" \
"       c[i] = a[i] + b[i];                                          \n" \
"}                                                                   \n" \
"\n";

//------------------------------------------------------------------------------


int main(int argc, char** argv)
{
    int             err;               // error code returned from OpenCL calls
    float           h_a[LENGTH];       // a vector
    float           h_b[LENGTH];       // b vector
    float           h_c[LENGTH];       // c vector (a+b) returned from the compute device
    unsigned int    correct;           // number of correct results

    size_t global;                     // global domain size

    cl_device_id     device_id;        // compute device id
    cl_context       context;          // compute context
    cl_command_queue commands;         // compute command queue
    cl_program       program;          // compute program
    cl_kernel        ko_vadd;          // compute kernel

    cl_mem d_a;                        // device memory used for the input  a vector
    cl_mem d_b;                        // device memory used for the input  b vector
    cl_mem d_c;                        // device memory used for the output c vector

    // Fill vectors a and b with random float values
    int i = 0;
    int count = LENGTH;
    for(i = 0; i < count; i++){
        h_a[i] = rand() / (float)RAND_MAX;
        h_b[i] = rand() / (float)RAND_MAX;
    }
```

```
// Set up platform and GPU device

cl_uint numPlatforms;

// Find number of platforms
err = clGetPlatformIDs(0, NULL, &numPlatforms);
if (err != CL_SUCCESS || numPlatforms <= 0)
{
    printf("Error: Failed to find a platform!\n%s\n",err_code(err));
    return EXIT_FAILURE;
}

// Get all platforms
cl_platform_id Platform[numPlatforms];
err = clGetPlatformIDs(numPlatforms, Platform, NULL);
if (err != CL_SUCCESS || numPlatforms <= 0)
{
    printf("Error: Failed to get the platform!\n%s\n",err_code(err));
    return EXIT_FAILURE;
}

// Secure a GPU
for (i = 0; i < numPlatforms; i++)
{
    err = clGetDeviceIDs(Platform[i], DEVICE, 1, &device_id, NULL);
    if (err == CL_SUCCESS)
    {
        break;
    }
}

if (device_id == NULL)
{
    printf("Error: Failed to create a device group!\n%s\n",err_code(err));
    return EXIT_FAILURE;
}

err = output_device_info(device_id);

// Create a compute context
context = clCreateContext(0, 1, &device_id, NULL, NULL, &err);
if (!context)
{
    printf("Error: Failed to create a compute context!\n%s\n", err_code(err));
    return EXIT_FAILURE;
}

// Create a command queue
commands = clCreateCommandQueue(context, device_id, 0, &err);
if (!commands)
{
    printf("Error: Failed to create a command commands!\n%s\n", err_code(err));
    return EXIT_FAILURE;
}

// Create the compute program from the source buffer
program = clCreateProgramWithSource(context, 1, (const char **) & KernelSource, NULL,
    &err);
if (!program)
{
    printf("Error: Failed to create compute program!\n%s\n", err_code(err));
    return EXIT_FAILURE;
}
```

```
// Build the program
err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
if (err != CL_SUCCESS)
{
    size_t len;
    char buffer[2048];

    printf("Error: Failed to build program executable!\n%s\n", err_code(err));
    clGetProgramBuildInfo(program, device_id, CL_PROGRAM_BUILD_LOG, sizeof(buffer),
        buffer, &len);
    printf("%s\n", buffer);
    return EXIT_FAILURE;
}

// Create the compute kernel from the program
ko_vadd = clCreateKernel(program, "vadd", &err);
if (!ko_vadd || err != CL_SUCCESS)
{
    printf("Error: Failed to create compute kernel!\n%s\n", err_code(err));
    return EXIT_FAILURE;
}

// Create the input (a, b) and output (c) arrays in device memory
d_a  = clCreateBuffer(context,  CL_MEM_READ_ONLY,  sizeof(float) * count, NULL, NULL);
d_b  = clCreateBuffer(context,  CL_MEM_READ_ONLY,  sizeof(float) * count, NULL, NULL);
d_c  = clCreateBuffer(context,  CL_MEM_WRITE_ONLY, sizeof(float) * count, NULL, NULL);
if (!d_a || !d_b || !d_c)
{
    printf("Error: Failed to allocate device memory!\n");
    exit(1);
}

// Write a and b vectors into compute device memory
err = clEnqueueWriteBuffer(commands, d_a, CL_TRUE, 0, sizeof(float) * count, h_a, 0,
    NULL, NULL);
if (err != CL_SUCCESS)
{
    printf("Error: Failed to write h_a to source array!\n%s\n", err_code(err));
    exit(1);
}

err = clEnqueueWriteBuffer(commands, d_b, CL_TRUE, 0, sizeof(float) * count, h_b, 0,
    NULL, NULL);
if (err != CL_SUCCESS)
{
    printf("Error: Failed to write h_b to source array!\n%s\n", err_code(err));
    exit(1);
}

// Set the arguments to our compute kernel
err  = clSetKernelArg(ko_vadd, 0, sizeof(cl_mem), &d_a);
err |= clSetKernelArg(ko_vadd, 1, sizeof(cl_mem), &d_b);
err |= clSetKernelArg(ko_vadd, 2, sizeof(cl_mem), &d_c);
err |= clSetKernelArg(ko_vadd, 3, sizeof(unsigned int), &count);
if (err != CL_SUCCESS)
{
    printf("Error: Failed to set kernel arguments!\n");
    exit(1);
}

double rtime = wtime();

// Execute the kernel over the entire range of our 1d input data set
```

```
    // letting the OpenCL runtime choose the work-group size
    global = count;
    err = clEnqueueNDRangeKernel(commands, ko_vadd, 1, NULL, &global, NULL, 0, NULL, NULL)
        ;
    if (err)
    {
        printf("Error: Failed to execute kernel!\n%s\n", err_code(err));
        return EXIT_FAILURE;
    }

    // Wait for the commands to complete before stopping the timer
    clFinish(commands);
    rtime = wtime() - rtime;
    printf("\nThe kernel ran in %lf seconds\n",rtime);

    // Read back the results from the compute device
    err = clEnqueueReadBuffer( commands, d_c, CL_TRUE, 0, sizeof(float) * count, h_c, 0,
        NULL, NULL );
    if (err != CL_SUCCESS)
    {
        printf("Error: Failed to read output array!\n%s\n", err_code(err));
        exit(1);
    }

    // Test the results
    correct = 0;
    float tmp;

    for(i = 0; i < count; i++)
    {
        tmp = h_a[i] + h_b[i];      // assign element i of a+b to tmp
        tmp -= h_c[i];              // compute deviation of expected and output result
        if(tmp*tmp < TOL*TOL)         // correct if square deviation is less than tolerance
             squared
            correct++;
        else {
            printf(" tmp %f h_a %f h_b %f h_c %f \n",tmp, h_a[i], h_b[i], h_c[i]);
        }
    }

    // summarize results
    printf("C = A+B:  %d out of %d results were correct.\n", correct, count);

    // cleanup then shutdown
    clReleaseMemObject(d_a);
    clReleaseMemObject(d_b);
    clReleaseMemObject(d_c);
    clReleaseProgram(program);
    clReleaseKernel(ko_vadd);
    clReleaseCommandQueue(commands);
    clReleaseContext(context);

    return 0;
}
```

2.3 Vadd in C++

Here is the same program written in C++. The OpenCL kernel is provided in a separate file which is included by the compliler. This code shows the part of the program that will actually execute on the GPU. This also allows for reutilization of low level OpenCL code on future programs. A researcher would build up a library over time, making accelerated parallel programming easier.

The vadd.cl file containing the kernel

vaddCode/vadd.cl

```
//------------------------------------------------------------------------------
//
// kernel:  vadd
//
// Purpose: Compute the elementwise sum c = a+b
//
// input: a and b float vectors of length count
//
// output: c float vector of length count holding the sum a + b
//

__kernel void vadd(
   __global float* a,
   __global float* b,
   __global float* c,
   const unsigned int count)
{
   int i = get_global_id(0);
   if(i < count)  {
       c[i] = a[i] + b[i];
   }
}
```

The vadd.cpp file containing the program to be compiled

vaddCode/vadd.cpp

```
//------------------------------------------------------------------------------
//
// Name:        vadd_cpp.cpp
//
// Purpose:     Elementwise addition of two vectors (c = a + b)
//
//                      c = a + b
//
// HISTORY:     Written by Tim Mattson, June 2011
//              Ported to C++ Wrapper API by Benedict Gaster, September 2011
//              Updated to C++ Wrapper API v1.2 by Tom Deakin and Simon McIntosh-Smith,
    October 2012
//              Updated to C++ Wrapper v1.2.6 by Tom Deakin, August 2013
//
```

```
//------------------------------------------------------------------------------

#define __CL_ENABLE_EXCEPTIONS

#include "cl.hpp"

#include "util.hpp" // utility library

#include <vector>
#include <cstdio>
#include <cstdlib>
#include <string>

#include <iostream>
#include <fstream>

// pick up device type from compiler command line or from the default type
#ifndef DEVICE
#define DEVICE CL_DEVICE_TYPE_DEFAULT
#endif

char* err_code(cl_int);

//------------------------------------------------------------------------------

#define TOL    (0.001)   // tolerance used in floating point comparisons
#define LENGTH (1024)    // length of vectors a, b, and c

int main(void)
{
    std::vector<float> h_a(LENGTH);                 // a vector
    std::vector<float> h_b(LENGTH);                 // b vector
    std::vector<float> h_c (LENGTH, 0xdeadbeef);    // c = a + b, from compute device

    cl::Buffer d_a;                         // device memory used for the input  a vector
    cl::Buffer d_b;                         // device memory used for the input  b vector
    cl::Buffer d_c;                        // device memory used for the output c vector

    // Fill vectors a and b with random float values
    int count = LENGTH;
    for(int i = 0; i < count; i++)
    {
        h_a[i]  = rand() / (float)RAND_MAX;
        h_b[i]  = rand() / (float)RAND_MAX;
    }

    try
    {
        // Create a context
        cl::Context context(DEVICE);

        // Load in kernel source, creating a program object for the context

        cl::Program program(context, util::loadProgram("vadd.cl"), true);

        // Get the command queue
        cl::CommandQueue queue(context);

        // Create the kernel functor

        auto vadd = cl::make_kernel<cl::Buffer, cl::Buffer, cl::Buffer, int>(program, "
            vadd");

        d_a   = cl::Buffer(context, begin(h_a), end(h_a), true);
```

```
        d_b   = cl::Buffer(context, begin(h_b), end(h_b), true);

        d_c  = cl::Buffer(context, CL_MEM_WRITE_ONLY, sizeof(float) * LENGTH);

        util::Timer timer;

        vadd(
            cl::EnqueueArgs(
                queue,
                cl::NDRange(count)),
            d_a,
            d_b,
            d_c,
            count);

        queue.finish();

        double rtime = static_cast<double>(timer.getTimeMilliseconds()) / 1000.0;
        printf("\nThe kernels ran in %lf seconds\n", rtime);

        cl::copy(queue, d_c, begin(h_c), end(h_c));

        // Test the results
        int correct = 0;
        float tmp;
        for(int i = 0; i < count; i++) {
            tmp = h_a[i] + h_b[i]; // expected value for d_c[i]
            tmp -= h_c[i];                      // compute errors
            if(tmp*tmp < TOL*TOL) {      // correct if square deviation is less
                correct++;                          //  than tolerance squared
            }
            else {

                printf(
                    " tmp %f h_a %f h_b %f  h_c %f \n",
                    tmp,
                    h_a[i],
                    h_b[i],
                    h_c[i]);
            }
        }

        // summarize results
        printf(
            "vector add to find C = A+B:  %d out of %d results were correct.\n",
            correct,
            count);
    }
    catch (cl::Error err) {
        std::cout << "Exception\n";
        std::cerr
            << "ERROR: "
            << err.what()
            << "("
            << err_code(err.err())
           << ")"
           << std::endl;
    }
}
```

2.4 Vadd in Python

There is also a higher level framework that may be more approachable for researchers who are not computer scientists called PyOpenCL. PyOpenCL allows programs written in the scripting language Python to utilize the OpenCL extensions for the parallelizable portions of the computational model while remaining easier to read and write. The code is also somewhat more portable, and can be shared with other researchers so long as they have a working Python-2.7 implementation with numerical Python support (numpy). The PyOpenCL code to accomplish vector addition. (Note the reduction in coding required, and that the kernel code is included directly):

vaddCode/vadd.py

```
#
# Vadd
#
# Element wise addition of two vectors (c = a + b)
# Asks the user to select a device at runtime
#
# History: C version written by Tim Mattson, December 2009
#          C version Updated by Tom Deakin and Simon McIntosh-Smith, October 2012
#          Ported to Python by Tom Deakin, July 2013
#

# Import the Python OpenCL API
import pyopencl as cl
# Import the Python Maths Library (for vectors)
import numpy

# Import a library to print out the device information
import deviceinfo

# Import Standard Library to time the execution
from time import time
#------------------------------------------------------------------------------

# tolerance used in floating point comparisons
TOL = 0.001
# length of vectors a, b and c
LENGTH = 1024

#------------------------------------------------------------------------------
#
# Kernel: vadd
#
# To compute the elementwise sum c = a + b
#
# Input: a and b float vectors of length count
# Output c float vector of length count holding the sum a + b

kernelsource = """
```

```
__kernel void vadd(
    __global float* a,
    __global float* b,
    __global float* c,
    const unsigned int count)
{
    int i = get_global_id(0);
    if (i < count)
        c[i] = a[i] + b[i];
}
"""

#---------------------------------------------------------------------------

# Main procedure

# Create a compute context
# Ask the user to select a platform/device on the CLI
context = cl.create_some_context()

# Print out device info
deviceinfo.output_device_info(context.devices[0])

# Create a command queue
queue = cl.CommandQueue(context)

# Create the compute program from the source buffer
# and build it
program = cl.Program(context, kernelsource).build()

# Create a and b vectors and fill with random float values
h_a = numpy.random.rand(LENGTH).astype(numpy.float32)
h_b = numpy.random.rand(LENGTH).astype(numpy.float32)
# Create an empty c vector (a+b) to be returned from the compute device
h_c = numpy.empty(LENGTH).astype(numpy.float32)

# Create the input (a, b) arrays in device memory and copy data from host
d_a = cl.Buffer(context, cl.mem_flags.READ_ONLY | cl.mem_flags.COPY_HOST_PTR, hostbuf=h_a)
d_b = cl.Buffer(context, cl.mem_flags.READ_ONLY | cl.mem_flags.COPY_HOST_PTR, hostbuf=h_b)
# Create the output (c) array in device memory
d_c = cl.Buffer(context, cl.mem_flags.WRITE_ONLY, h_c.nbytes)

# Start the timer
rtime = time()

# Execute the kernel over the entire range of our 1d input
# allowing OpenCL runtime to select the work group items for the device
vadd = program.vadd
vadd.set_scalar_arg_dtypes([None, None, None, numpy.uint32])
vadd(queue, h_a.shape, None, d_a, d_b, d_c, LENGTH)

# Wait for the commands to finish before reading back
queue.finish()
rtime = time() - rtime
print "The kernel ran in", rtime, "seconds"

# Read back the results from the compute device
cl.enqueue_copy(queue, h_c, d_c)

# Test the results
correct = 0;
for a, b, c in zip(h_a, h_b, h_c):
    # assign element i of a+b to tmp
    tmp = a + b
```

```
    # compute the deviation of expected and output result
    tmp -= c
    # correct if square deviation is less than tolerance squared
    if tmp*tmp < TOL*TOL:
        correct += 1
    else:
        print "tmp", tmp, "h_a", a, "h_b", b, "h_c", c

# Summarize results
print "C = A+B:", correct, "out of", LENGTH, "results were correct."
```

Although Python is more considered easier to learn for the someone new to programming, the following test show that the speed increases available with the C++ code are significant enough that we will not consider it further.

2.5 Results

Running the C and Python versions of the parallel vector addition 10 times each on the APU test machine produces average run times of 0.47 milliseconds for the C code and 1.25 milliseconds for the Python code with the default vector length of 1024. That is each program run, vectors with 1024 random 32 bit floating point elements each were added together and the results compared to the expected result by doing the adddition on the CPU. So the C code was significantly faster at the cost of considerable programming overhead.

An interesting result occured when trying to time the compiled C++ version. The default vector length of 1024 failed to produce any measured time in milliseconds. The computation didn't take long enough to register a time change in the getTime() function probably due to insufficient resolution in the timer function. Changing the vector length to 16,777,216 (2^{24}) produced a time of 71 milliseconds. The C++ compiler on the test machine would appear to access to speed increases beyond what a non-expert programmer could hope to achieve. This is not the expected behavior for C vs. C++ code. We believe the result is due to the C++ compiler performaing optimizations using the installed high performance BLAS and LAPACK libraries when compiling C++ but not C code. Since the C++ wrapper available

for OpenCL is sufficiently accessable, i.e. it does not require explicit memory management on the part of the programmer, we do not consider the PyOpenCL framework further. It is still useful to the Python programmer who needs to accelerate small portions of code in larger models.

It should be noted that the C++ OpenCL code executed faster on the CPU cores of the AMD APU (29 ms) than it did on the GPU cores. We would conclude that for simple programs the higher clock speed of the CPU and the built in SIMD (single instruction multi-data) vector processing units can eliminate the need for moving computations to the GPU. We will see in the next chapter that for larger matrix operations there is a distinct advantage.

Image Rotation

To illustrate using the C++ framework with a simple example that has accessible source code, we will look at an image rotation from Chapter 4 of "Heterogeneous Computing with OpenCL" [5]. Rotating an image is a mathematically simple change of coordinates for each individual pixel. To rotate a point (x_1, y_1) around a point (x_0, y_0) by an angle θ, we find the new coordinates (x_2, y_2) by:

$$x_2 = \cos(\theta)(x_1 - x_0) + \sin(\theta)(y_1 - y_0)$$
$$y_2 = -\sin(\theta)(x_1 - x_0) + \cos(\theta)(y_1 - y_0)$$

If we are rotating around the origin $(0,0)$, we have

$$x_2 = \cos(\theta)(x_1) + \sin(\theta)(y_1)$$
$$y_2 = -\sin(\theta)(x_1) + \cos(\theta)(y_1)$$

To do this in parallel on the graphics card requires a bit more work.

3.1 Setting up the environment in C++

The source code in Appendix A is divided into the C++ file that will be given to the compiler along with another file that contains functions for working with bitmap images. The C++ source has the compiler include the OpenCL specific funtions in the separate "rotation.cl" file. The source code is well commented and not included here for readability. The program

first checks for available OpenCL capable devices using functions in the OpenCL code library cl.hpp. After a device is chosen, memory is allocated, input and output files are declared, and then the OpenCL kernel is compiled.

3.2 Setting up the rotation in OpenCL

The OpenCL kernel code from rotation.cl is compilied. Floating point arrays are declared to hold the source and destination data. This allocates memory for the data structures. Individual (x, y) pairs are indexed and sent for computation in parallel. After rotation, the new coordinates are checked to ensure they are still in the image.

3.3 Results

After rotation, the pixels that remain wthin the image boundaries are written to the output file.

This example demonstrates the amount of programming necessary to perform a simple task using OpenCL to explicitly address the graphics hardware. This should be compared to high level scripted languages that could include a single "rotateImage(image.jpg,θ)" command. It is provided here as an introduction to the language. Next we will look at a more computationally intensive application.

Singular Value Decomposition

4.1 Definition

Singular value decomposition (SVD) is one type of operation that lends itself readily to parallelization. For example, SVD is used in Latent Semantic Indexing (LSI), a concept based indexing method for text documents. LSI is used to construct a vector space in which the collection of text documents is represented by a *term-document* matrix $A = (a_{ij}) \in \mathbb{R}^{m \times n}$ where a_{ij} represents the number of times the term i appears in the document j, m is the number of terms, and n is the number of documents. Each document is then a column vector in the matrix. Searching for information in the collection requires constructing another column vector from the search terms and returning the column vectors closest to the search vector. LSI is a technique used in search engines [12]. To compute the SVD of a matrix A we need to find U, S, V, such that

$$A = USV^T,$$

where U is an $m \times m$ matrix composed of the eigenvectors of the matrix AA^T, V is an $n \times n$ matrix composed of the eigenvectors of A^TA, and S is an $m \times n$ diagonal matrix composed of the eigenvalues of A.

4.2 Householder bidiagonalization

Using Gram-Schmidt orthogonalization on a computer is susceptible to problems related to machine division with arbitrarily small numbers [2]. In order to avoid this we can use

Householder transformations to get a bidiagonalization of A prior to decompostition. We can express reflections across a plane orthogonal to a unit normal vector v as:

$$H = I - 2vv^T$$

For a column vector x in S and unit vector $y = e_i$ (in the normal basis), we take the reflection throught the hyperplane that bisects the angle between x and y by taking the hyperplane normal to $x - \|x\|y$. That is, lettting $u = x - \|x\|y$ and $v = u/\|u\|$, we have

$$(I - 2vv^T)x = x - 2\frac{(x + \|x\|y)(x^T x + \|x\|x^T y)}{\|x\|^2 + 2x^T y\|x\| + \|x\|^2\|y\|^2}$$

Or, $x - (x - \|x\|y) = \|x\|y$. Since $y = e_1$, this zeros out all but the first entry of x. With similar choices for the other column vectors, we can get a transformation matrix that zeros out the entries below the diagonal. And a series of such transformations are used to get the bidiagonal form required. This step can be done in parallel for the columns of A.

4.3 Results

Using the source code in Appendix B to compare the execution time of a parallel algorithm to a sequential one see Figure 4.1, we find the following results on the Nvidia machine for bidiagonalization of a random matrix $A_{m,n}$ and given values of m and n we list the execution time in seconds for the parallel version of the bidiagonalization routine (par) and the sequential (seq):

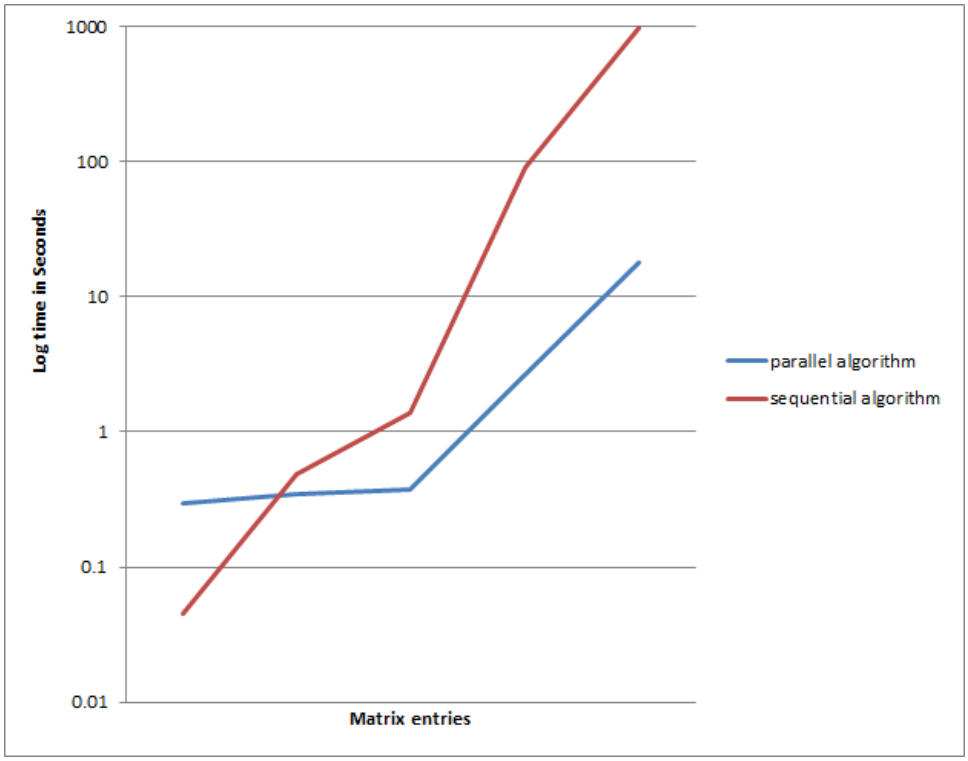

Figure 4.1: Observed time required for transform

Matrix Dimensions						
m	256	512	256	2048	2048	4096
n	256	512	2048	256	2048	4096
Execution time in seconds						
par	0.298	0.346	0.373	0.370	2.668	17.870
seq	0.045	0.492	1.930	1.391	91.383	978.205

It is clear, that at small matrix dimensions the sequential algorithm is faster. This is often due to the faster speed of the CPU vector units compared to the GPU. But as the size of

A increases, the parallel algorithm clearly comes to dominate. At 16 million entries, the difference is 17 seconds vs. 16 minutes. For any model that requires iteratively factoring large matrices it would be well worth the programming overheard to move the computation to the graphics hardware.

Conclusion

We have shown that significant acceleration using graphics hardware and the OpenCL framework is possible although not without some programming effort. Researcher should remember that CPUs are faster than GPUs for small work group sizes but discrete GPUs are significantly faster on large workloads. It should be noted that neither CPUs or GPUs can accelerate computations without access to data. Researchers can achieve significant speed increases without programming effort from first adding more memory and high speed solid state disk drives to their systems.

In the near future researchers will have even more hardware acceleration possibiliites. Driven largely by consumer demand for more powerful portable devices such as tablets and smartphones, CPU manufacturers are designing systems with more specialized capabilities directly on the CPU hardware. A description of this effort is available from the HSA(Heterogeneous Systems Architecture) foundation web site [4]:

> The Heterogeneous System Architecture (HSA) is designed to efficiently support a wide assortment of data-parallel and task-parallel programming models.
> A single HSA system can support multiple instruction sets based on latency and throughput compute units.
> HSA supports two machine models: large mode (64-bit address space) and small mode (32-bit address space).
> An HSA-compliant system will meet the requirements for a queuing model, a

memory model, quality of service, and an instruction set for parallel processing and enabling heterogeneous programming models for computing platforms using standardized interfaces, processes, communication protocols, and memory models.

HSA is an optimized platform that will be leveraged to enhance the OpenCL runtime. HSA is not an alternative to OpenCL. HSA benefits OpenCL by removing memory copies, bringing low latency dispatch, and improving memory model and pointers shared between the CPU and GPU. The HSA Foundation will also work with all software foundations and open initiatives for the promotion of respective language runtimes and libraries around the advancement of heterogeneous computing.

References

[1] TOP500 Supercomputer Sites. http://www.top500.org, November 2013.

[2] Bindel. Householder transformations. http://http://www.cs.cornell.edu/~bindel/class/cs6210-f12/notes/lec16.pdf, November 2013.

[3] Oak Ridge Leadership Computing Facility. Introducing Titan. http://olcf.ornl.gov/titan, November 2013.

[4] HSA Foundation. The hsa foundation. http://hsafoundation.com, November 2013.

[5] Benedict Gaster, Lee Howes, David R. Kaeli, Perhaad Mistry, and Dana Schaa. *Heterogeneous Computing with OpenCL*. Burlington : Elsevier Science, Burlington, 2011.

[6] Andreas Grothey. Financial applications: Parallel portfolio optimization. In Trobec et al. [14], pages 435–470.

[7] Pascal Hénon, Pierre Ramet, and Jean Roman. On using an hybrid MPI-thread programming for the implementation of a parallel sparse direct solver on a network of SMP nodes. In Wyrzykowski et al. [16], pages 1050–1057.

[8] Andreas Klöckner, Nicolas Pinto, Yunsup Lee, B. Catanzaro, Paul Ivanov, and Ahmed Fasih. PyCUDA and PyOpenCL: A Scripting-Based Approach to GPU Run-Time Code Generation. *Parallel Computing*, 38(3):157–174, 2012.

[9] Rade Kutil. Short-vector SIMD parallelization in signal processing. In Trobec et al. [14], pages 397–434.

[10] Simon McIntosh-Smith and Tom Deakin. Hands-on OpenCL. http://handsonopencl.github.io/, November 2013.

[11] Rishiyur S. Nikhil. *Implicit parallel programming in pH*. San Francisco : Morgan Kaufmann Publishers, San Francisco, 2001.

[12] Gabriel Okša and Marián Vajteršic. Parallel SVD computing in the latent semantic indexing applications for data retrieval. In Trobec et al. [14], pages 359–396.

[13] Bhushan Dattatraya Rayrikar. Implemetation of the singular value decomposition using OpenCL. Master's thesis, Clemson University, December 2011.

[14] Roman Trobec, Marián Vajteršic, and Peter Zinterhof, editors. *Parallel computing : numerics, applications, and trends*. Dordrecht ; New York : Springer, Dordrecht ; New York, 2009.

[15] Nathan Whitehead and Alex Fit-Florea. Precision and performance: Floating point and IEEE 754 compliance for NVIDIA GPUS. `http://docs.nvidia.com/cuda/pdf/Floating_Point_on_NVIDIA_GPU.pdf`, July 2013.

[16] Roman Wyrzykowski, Jack Dongarra, Norbert Meyer, and Jerzy Waśniewski, editors. *Parallel processing and applied mathematics : 6th international conference, PPAM 2005, Poznań, Poland, September 11-14, 2005 : revised selected papers*. Berlin ; New York : Springer, Berlin ; New York, 2006.

Appendix A

Code Listing Image Rotation

The kernel code to rotate each individual pixel:

rotateCode/rotation.cl

```
__kernel
void img_rotate(__global float* dest_data,
                __global float* src_data,
                         int W,
                         int H,
                       float sinTheta,
                       float cosTheta) {

   //Work-item gets its index within index space
   const int ix = get_global_id(0);
   const int iy = get_global_id(1);

   //Calculate location of data to move into (ix,iy)
   //Output decomposition as mentioned
   float x0 = W/2.0f;
   float y0 = H/2.0f;

   float xOff = ix - x0;
   float yOff = iy - y0;

   int xpos = (int)(xOff*cosTheta + yOff*sinTheta + x0 );
   int ypos = (int)(yOff*cosTheta - xOff*sinTheta + y0 );

   // Bounds Checking
   if((xpos>=0) && (xpos< W) && (ypos>=0) && (ypos< H)) {

      // Read (ix,iy) src_data and store at (xpos,ypos) in
      // dest_data
      // In this case, because we rotating about the origin
      // and there is no translation, we know that (xpos,ypos)
      // will be unique for each input (ix,iy) and so each
      // work-item can write its results independently

      dest_data[iy*W+ix] = src_data[ypos*W+xpos];
   }
}
```

The c++ code to use the kernel code to perform the rotations:

rotateCode/rotation.cpp

```
#define __NO_STD_VECTOR
```

```
#define __CL_ENABLE_EXCEPTIONS


#include "CL/cl.hpp"
#include <utility>
#include <iostream>
#include <fstream>
#include <math.h>
#include <string>

//BMP utilities
#include "bmpfuncs.h"

//! You will need to tweak these 2 parameters
//! Using 0 will always choose the 1st implementation found
#define PLATFORM_TO_USE 0
#define DEVICE_TYPE_TO_USE  CL_DEVICE_TYPE_CPU

//using namespace cl;
int main(int argc, char ** argv)
{

try {

    cl_int err;
    cl::vector<cl::Platform> platforms;
    cl::Platform::get(&platforms);

    std::cout << "Number of platforms:\t" << platforms.size() << std::endl;
    for (cl::vector<cl::Platform>::iterator i = platforms.begin(); i != platforms.end();
        ++i) {
        // pick a platform and do something
        std::cout << " Platform Name: " << (*i).getInfo<CL_PLATFORM_NAME>().c_str()<< std
            ::endl;
    }

    float theta = 3.14159/6;
    int W;
    int H;

    const char* inputFile = "input.bmp";
    const char* outputFile = "output.bmp";

    // Homegrown function to read a BMP from file
    float* ip = readImage(inputFile, &W, &H);
    float * op = new float[W*H];


    //Lets choose the first platform
    cl_context_properties cps[3] = {


    CL_CONTEXT_PLATFORM, (cl_context_properties)(platforms[PLATFORM_TO_USE])(), 0};

    // Select the default platform and create a context
    // using this platform for a GPU type device

    cl::Context context(DEVICE_TYPE_TO_USE, cps);

    cl::vector<cl::Device> devices = context.getInfo<CL_CONTEXT_DEVICES>();

    //Lets create a command queue on the first device
    cl::CommandQueue queue = cl::CommandQueue(context, devices[0], 0, &err);
```

```
    //[H3] Step2 âĂŞ Declare Buffers and Move Data

    //We assume that the input image is the array âĂIJipâĂİ
    //and the angle of rotation is theta
    float cos_theta = cos(theta);
    float sin_theta = sin(theta);

    cl::Buffer d_ip = cl::Buffer(context, CL_MEM_READ_ONLY, W*H* sizeof(float));
    cl::Buffer d_op = cl::Buffer(context, CL_MEM_READ_WRITE, W*H* sizeof(float));
    queue.enqueueWriteBuffer(d_ip, CL_TRUE, 0, W*H* sizeof(float), ip);

    //[H3]Step3 âĂŞ Runtime kernel compilation

    std::ifstream sourceFileName("rotation.cl");


    std::string sourceFile(
                    std::istreambuf_iterator<char>( sourceFileName),
                     (std::istreambuf_iterator<char>())
                    );

    //std::cout<<sourceFile;

    cl::Program::Sources rotn_source(1,
            std::make_pair(sourceFile.c_str(),
                           sourceFile.length() +1
                           )
                       );

    cl::Program rotn_program(context, rotn_source);
    rotn_program.build(devices);

    cl::Kernel rotn_kernel(rotn_program, "img_rotate", &err);

    //[H3]Step4 âĂŞ Run the program
    rotn_kernel.setArg(0, d_op);
    rotn_kernel.setArg(1, d_ip);
    rotn_kernel.setArg(2, W);
    rotn_kernel.setArg(3, H);
    rotn_kernel.setArg(4, sin_theta);
    rotn_kernel.setArg(5, cos_theta);

    // Run the kernel on specific ND range
    cl::NDRange globalws(W,H);
    //In this example the local work group size is not important because
    //there is no communication between local work items

    queue.enqueueNDRangeKernel(rotn_kernel, cl::NullRange, globalws, cl::NullRange);
     //[H3]Step5 âĂŞ Read result back to host
    // Read buffer d_op into a local op array
    queue.enqueueReadBuffer(d_op, CL_TRUE, 0, W*H*sizeof(float), op);

    storeImage(op, outputFile, H, W, inputFile);

}
catch(cl::Error err)
{
   std::cout << err.what() << "(" << err.err() << ")" << std::endl;
}

}
```

The header files and c code for the functions called from the c++ program:

rotateCode/bmpfuncs.h

```
#ifndef __BMPFUNCS__
#define __BMPFUNCS__

typedef unsigned char uchar;

float* readImage(const char *filename, int* widthOut, int* heightOut);
void storeImage(float *imageOut, const char *filename, int rows, int cols,
                const char* refFilename);

#endif
```

rotateCode/bmpfuncs.c

```
#include <stdio.h>
#include <stdlib.h>

#include "bmpfuncs.h"

void storeImage(float *imageOut, const char *filename, int rows, int cols,
                const char* refFilename) {

   FILE *ifp, *ofp;
   unsigned char tmp;
   int offset;
   unsigned char *buffer;
   int i, j;

   int bytes;

   int height, width;

   ifp = fopen(refFilename, "rb");
   if(ifp == NULL) {
      perror(filename);
      exit(-1);
   }

   fseek(ifp, 10, SEEK_SET);
   fread(&offset, 4, 1, ifp);

   fseek(ifp, 18, SEEK_SET);
   fread(&width, 4, 1, ifp);
   fread(&height, 4, 1, ifp);

   fseek(ifp, 0, SEEK_SET);

   buffer = (unsigned char *)malloc(offset);
   if(buffer == NULL) {
      perror("malloc");
      exit(-1);
   }

   fread(buffer, 1, offset, ifp);

   printf("Writing output image to %s\n", filename);
   ofp = fopen(filename, "wb");
   if(ofp == NULL) {
      perror("opening output file");
      exit(-1);
```

```
    }
    bytes = fwrite(buffer, 1, offset, ofp);
    if(bytes != offset) {
       printf("error writing header!\n");
       exit(-1);
    }

    // NOTE bmp formats store data in reverse raster order (see comment in
    // readImage function), so we need to flip it upside down here.
    int mod = width % 4;
    if(mod != 0) {
       mod = 4 - mod;
    }
    //    printf("mod = %d\n", mod);
    for(i = height-1; i >= 0; i--) {
       for(j = 0; j < width; j++) {
          tmp = (unsigned char)imageOut[i*cols+j];
          fwrite(&tmp, sizeof(char), 1, ofp);
       }
       // In bmp format, rows must be a multiple of 4-bytes.
       // So if we're not at a multiple of 4, add junk padding.
       for(j = 0; j < mod; j++) {
          fwrite(&tmp, sizeof(char), 1, ofp);
       }
    }

    fclose(ofp);
    fclose(ifp);

    free(buffer);
}

/*
 * Read bmp image and convert to byte array. Also output the width and height
 */
float* readImage(const char *filename, int* widthOut, int* heightOut) {

    uchar* imageData;

    int height, width;
    uchar tmp;
    int offset;
    int i, j;

    printf("Reading input image from %s\n", filename);
    FILE *fp = fopen(filename, "rb");
    if(fp == NULL) {
        perror(filename);
        exit(-1);
    }

    fseek(fp, 10, SEEK_SET);
    fread(&offset, 4, 1, fp);

    fseek(fp, 18, SEEK_SET);
    fread(&width, 4, 1, fp);
    fread(&height, 4, 1, fp);

    printf("width = %d\n", width);
    printf("height = %d\n", height);

    *widthOut = width;
    *heightOut = height;
```

```
imageData = (uchar*)malloc(width*height);
if(imageData == NULL) {
    perror("malloc");
    exit(-1);
}

fseek(fp, offset, SEEK_SET);
fflush(NULL);

int mod = width % 4;
if(mod != 0) {
    mod = 4 - mod;
}

// NOTE bitmaps are stored in upside-down raster order.  So we begin
// reading from the bottom left pixel, then going from left-to-right,
// read from the bottom to the top of the image.  For image analysis,
// we want the image to be right-side up, so we'll modify it here.

// First we read the image in upside-down

// Read in the actual image
for(i = 0; i < height; i++) {

   // add actual data to the image
   for(j = 0; j < width; j++) {
      fread(&tmp, sizeof(char), 1, fp);
      imageData[i*width + j] = tmp;
   }
   // For the bmp format, each row has to be a multiple of 4,
   // so I need to read in the junk data and throw it away
   for(j = 0; j < mod; j++) {
      fread(&tmp, sizeof(char), 1, fp);
   }
}

// Then we flip it over
int flipRow;
for(i = 0; i < height/2; i++) {
   flipRow = height - (i+1);
   for(j = 0; j < width; j++) {
      tmp = imageData[i*width+j];
      imageData[i*width+j] = imageData[flipRow*width+j];
      imageData[flipRow*width+j] = tmp;
   }
}

fclose(fp);

// Input image on the host
float* floatImage = NULL;
floatImage = (float*)malloc(sizeof(float)*width*height);
if(floatImage == NULL) {
   perror("malloc");
   exit(-1);
}

// Convert the BMP image to float (not required)
for(i = 0; i < height; i++) {
   for(j = 0; j < width; j++) {
      floatImage[i*width+j] = (float)imageData[i*width+j];
   }
}
```

```
    free(imageData);
    return floatImage;
}
```

Appendix B

Code Listing Singular Value Decompostion

bidiagCode/bidiag.h

```
/**************************************************************************
 *
 * Copyright (C) 2012 Travis Askham
 *
 * Permission is hereby granted, free of charge, to any person obtaining
 * a copy of this software and associated documentation files (the
 * "Software"), to deal in the Software without restriction, including
 * without limitation the rights to use, copy, modify, merge, publish,
 * distribute, sublicense, and/or sell copies of the Software, and to
 * permit persons to whom the Software is furnished to do so, subject to
 * the following conditions:
 *
 * The above copyright notice and this permission notice shall be
 * included in all copies or substantial portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
 * EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
 * MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
 * NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS
 * BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN
 * ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN
 * CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
 * SOFTWARE.
 *
**************************************************************************/

#ifndef BIDIAG
#define BIDIAG

void bidiag_seq(int m, int n, double *restrict A, double *restrict alpha,
                        double *restrict beta);

void left_householder(int m, int n, int l, int k, double *restrict v,
                                                        double *restrict A);

void right_householder(int m, int n, int l, int k, double *restrict v,
                                                        double *restrict A);

void form_u(int m, int n, const double*restrict A_mod, double*restrict U);
void form_v(int m, int n, const double*restrict A_mod, double*restrict V);

#endif
```

bidiagCode/bidiag_par.h

```
/**************************************************************************
```

```
 * Copyright (C) 2012 Travis Askham, Steven Delong
 * Permission is hereby granted, free of charge, to any person obtaining
 * a copy of this software and associated documentation files (the
 * "Software"), to deal in the Software without restriction, including
 * without limitation the rights to use, copy, modify, merge, publish,
 * distribute, sublicense, and/or sell copies of the Software, and to
 * permit persons to whom the Software is furnished to do so, subject to
 * the following conditions:
 *
 * The above copyright notice and this permission notice shall be
 * included in all copies or substantial portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
 * EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
 * MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
 * NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS
 * BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN
 * ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN
 * CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
 * SOFTWARE.
*****************************************************************/
#ifndef BIDIAG_PAR
#define BIDIAG_PAR
void bidiag_par(int m, int n, double *restrict A, double *restrict alpha,
                        double *restrict beta);

void form_u_par(int m, int n, const double*restrict A_mod, double*restrict U);
void form_v_par(int m, int n, const double*restrict A_mod, double*restrict V);

void find_reflector_and_scale_col(int m, int n, int mn, int col,
                                  int work_per_item_red, int wgdim_red, int
                                      work_per_item_upd, int wgdim_upd,
                                  cl_context ctx, cl_command_queue queue,
                                  cl_kernel knl_numsq_matcol,cl_kernel
                                      knl_update_scale_matcol,
                                  cl_kernel knl_sum,
                                  cl_mem buf_A, cl_mem buf_alpha,
                                  cl_mem buf_scratch, cl_mem buf_scratch2, cl_mem buf_temp);

void find_reflector_and_scale_row(int m, int n, int mn, int row,
                                  int work_per_item_red, int wgdim_red, int
                                      work_per_item_upd, int wgdim_upd,
                                  cl_context ctx, cl_command_queue queue,
                                  cl_kernel knl_normsq_matrow,cl_kernel
                                      knl_update_scale_matrow,
                                  cl_kernel knl_sum,
                                  cl_mem buf_A, cl_mem buf_beta,
                                  cl_mem buf_scratch, cl_mem buf_scratch2, cl_mem buf_temp);

void no_reflect( int m, int n, int mat_loc, int vec_loc,
                                  int work_per_item_nr, int wgdim_nr,
                                  cl_context ctx, cl_command_queue queue,
                                  cl_kernel knl_sanders,
                                  cl_mem buf_mat, cl_mem buf_vec);

void left_householder_par_2d(int m, int n, int mn, int offset,
                                  size_t ldim_dp[], size_t ldim_update_mat[],
                                  cl_context ctx, cl_command_queue queue,
                                  cl_kernel knl_left_dotprods, cl_kernel knl_update_left_hh,
                                  cl_mem buf_A, cl_mem buf_scratch);

void right_householder_par_2d(int m, int n, int mn, int offset,
                                  size_t ldim_dp[], size_t ldim_update_mat[],
                                  cl_context ctx, cl_command_queue queue,
```

```
                                cl_kernel knl_right_dotprods, cl_kernel
                                    knl_right_update_mat,
                                cl_mem buf_A, cl_mem buf_scratch);

void multU(int m,int n, int vecnum, double*restrict A_mod, double*restrict Y,double*
    restrict U);
void multV(int m,int n, int vecnum, double*restrict A_mod, double*restrict X,double*
    restrict V);
#endif
```

bidiagCode/bidiag.c

```
/*************************************************************************
 *
 * Copyright (C) 2012 Travis Askham
 *
 * Permission is hereby granted, free of charge, to any person obtaining
 * a copy of this software and associated documentation files (the
 * "Software"), to deal in the Software without restriction, including
 * without limitation the rights to use, copy, modify, merge, publish,
 * distribute, sublicense, and/or sell copies of the Software, and to
 * permit persons to whom the Software is furnished to do so, subject to
 * the following conditions:
 *
 * The above copyright notice and this permission notice shall be
 * included in all copies or substantial portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
 * EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
 * MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
 * NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS
 * BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN
 * ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN
 * CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
 * SOFTWARE.
 *
*************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "matrix_helper.h"
#include "bidiag.h"

void bidiag_seq(int m, int n, double*restrict A, double*restrict alpha, double*restrict
    beta){

/*********************************************************
 * Function: bidiag_seq
 *
 * Original Author: Travis Askham (12/20/2012)
 *
 * Description: reduces the input matrix to bidiagonal form
 * using a sequential algorithm. The reduction is accomplished via
 * orthogonal transformations.
 *
 * Input
 *
 *      m - the number of rows in the matrix A
 *      n - the number of columns in the matrix A
 *      A - the matrix, given in column major order
 *
 * Output
 *
```

```
*       alpha - the diagonal of the resulting bidiagonal matrix
*                       its length is min(m,n)
*       beta - the superdiagonal of the resulting bidiagonal matrix
*                       its length is min(m,n)-1 if n <= m or min(m,n) if n > m
*       A - has been overwritten with the information necessary to
*               reconstruct the orthogonal transformations
*
* ******************************************************/

        int mn;
        int sign;
        double temp_norm;
        double x1;

        if (m<n){
                mn = m;
        }
        else{
                mn = n;
        }

        for (int i=0; i< mn-1; i++){
                // do the left Householder reflection
                x1 = A[i+i*m];
                if ( x1 < 0){
                        sign = -1;
                }
                else{
                        sign = 1;
                }
                temp_norm = l2_normv(m-i,A+i+i*m);
                // norm of the partial column
                // starting at A[i,i]
                alpha[i] = -sign*temp_norm;
                A[i+i*m] += sign*temp_norm;// the unscaled reflector is
                // in the partial column
                // starting at A[i,i]
                // norm of reflector
                temp_norm = sqrt(2.0)*sqrt(temp_norm*temp_norm + fabs(temp_norm*x1));
                scale_vector(m-i,A+i+i*m,1.0/temp_norm);// scale the reflector
                left_householder(m,n,m-i,n-i-1,A+i+i*m,A+i+(i+1)*m); //apply reflection
                // to the remainder of the matrix
                if ( i < n-2){
                        // do the right Householder reflection
                        x1 = A[i+(i+1)*m];
                        if ( x1 < 0){
                                sign = -1;
                        }
                        else{
                                sign = 1;
                        }
                        temp_norm = l2_norm_mat_row(m,n,n-i-1,A+i+(i+1)*m);
                        // norm of the partial row
                        // starting at A[i,i+1]
                        beta[i] = -sign*temp_norm;
                        A[i+(i+1)*m] += sign*temp_norm;
                        // the unscaled reflector is
                        // starting at A[i,i+1]
                        // norm of reflector
                        temp_norm = sqrt(2.0)*sqrt(temp_norm*temp_norm + fabs(temp_norm*x1
                            ));
                        scale_mat_row(m,n,n-i-1,A+i+(i+1)*m,1.0/temp_norm);
                        // scale the reflector
                        right_householder(m,n,m-i-1,n-i-1,A+i+(i+1)*m,A+i+1+(i+1)*m);
```

```
                        //apply reflection
                         // to the remainder of the matrix
                }
                else{
                        // no reflection on right
                        beta[i] = A[i+(i+1)*m];
                        A[i+(i+1)*m] = 0;
                }
        }
        if ( n >= m+1 ){
                // do the right Householder reflection
                x1 = A[mn-1+mn*m];
                if ( x1 < 0){
                        sign = -1;
                }
                else{
                        sign = 1;
                }

                temp_norm = l2_norm_mat_row(m,n,n-mn,A+mn-1+mn*m);
                // norm of the partial row
                // starting at A[i,i+1]
                beta[mn-1] = -sign*temp_norm;
                A[mn-1+mn*m] += sign*temp_norm;
                // the unscaled reflector is
                // in the partial row
                // starting at A[i,i+1]
                // norm of reflector
                temp_norm = sqrt(2.0)*sqrt(temp_norm*temp_norm + fabs(temp_norm*x1));
                scale_mat_row(m,n,n-mn,A+mn-1+mn*m,1.0/temp_norm);
                // scale the reflector
                // no reflection on left
                alpha[mn-1] = A[mn-1+(mn-1)*m];
                A[mn-1 + (mn-1)*m] = 0;
        }
        else {
                // do the left Householder reflection
                x1 = A[mn-1+(mn-1)*m];
                if ( x1 < 0){
                        sign = -1;
                }
                else{
                        sign = 1;
                }

                temp_norm = l2_normv(m-mn+1,A+mn-1+(mn-1)*m);
                // norm of the partial column
                // starting at A[i,i]
                alpha[mn-1] = -sign*temp_norm;

                A[mn-1+(mn-1)*m] += sign*temp_norm;
                // the unscaled reflector is
                // in the partial column
                // starting at A[i,i]
                // norm of reflector
                temp_norm = sqrt(2.0)*sqrt(temp_norm*temp_norm + fabs(temp_norm*x1));
                scale_vector(m-mn+1,A+mn-1+(mn-1)*m,1.0/temp_norm);
                // scale the reflector
        }
        return;
}
void left_householder(int m, int n, int l, int k, double*restrict v, double*restrict A){

/*******************************************************
```

```
 * Function: left_householder
 *
 * Original Author: Travis Askham (12/20/2012)
 *
 * Input:
 *      m - number of rows in matrix
 *      n - number of columns in matrix
 *      l - number of rows in submatrix
 *      k - number of columns in submatrix
 *  v - reflection vector of length l
 *      A - pointer to top left of submatrix
 *
 *
 ************************************************************/

        double inner_prod;

        // step through columns
        for (int j=0; j<k; j++){
                inner_prod = dot_prod(l,v,A+j*m);
                for (int i=0; i<l; i++){
                        A[i+j*m] -= 2*v[i]*inner_prod;
                }
        }

        return;

}

void right_householder(int m, int n, int l, int k, double*restrict v, double*restrict A){

/************************************************************
 * Function: right_householder
 *
 * Original Author: Travis Askham (12/20/2012)
 *
 * Input:
 *      m - number of rows in matrix
 *      n - number of columns in matrix
 *      l - number of rows in submatrix
 *      k - number of columns in submatrix
 *  v - reflection row of length l (embedded in mxn matrix)
 *      A - pointer to top left of submatrix
 *
 *
 ************************************************************/

        double inner_prod;

        // step through columns
        for (int i=0; i<l; i++){
                inner_prod = dot_prod_mat_rows(m,n,k,v,A+i);
                for (int j=0; j<k; j++){
                        A[i+j*m] -= 2*v[j*m]*inner_prod;
                }
        }

        return;

}

void form_u(int m, int n, const double*restrict A_mod, double*restrict U){

/*******************************************************
```

```
 * Function: form_u
 *
 * Original Author: Travis Askham (12/20/2012)
 *
 * Input:
 *      m - number of rows in matrix
 *      n - number of columns in matrix
 *      A_mod - pointer to top left of matrix storing reflection vectors (comes from
 *  the bidiag_seq routine)
 *
 * Output:
 *      U - the left orthogonal matrix in the bidiagonal decomposition
 *
 *                                  A = U B V^T
 *
 * where B is bidiagonal
 *
 ************************************************************/

        int mn;
        int j_start;
        double inner_prod;
        if (m<n){
                mn = m;
        }
        else{
                mn = n;
        }

        // set the ith column of U to Ue_i
        for (int i=0; i < m; i++){
                set_vec_to_zero(m,U+i*m);
                U[i+i*m] = 1;
                if (i < mn-1 ){
                        j_start = i;
                }
                else{
                        j_start = mn-1;
                }
                for (int j=j_start; j >= 0; j--){
                        inner_prod = dot_prod(m-j,U+j+i*m,A_mod+j+j*m);
                        for (int k=j; k<m; k++){
                                U[k+i*m] -= 2*A_mod[k+j*m]*inner_prod;
                        }
                }
        }

        return;
}

void form_v(int m, int n, const double*restrict A_mod, double*restrict V){

/************************************************************
 * Function: form_v
 *
 * Original Author: Travis Askham (12/20/2012)
 *
 * Input:
 *      m - number of rows in matrix
 *      n - number of columns in matrix
 *      A_mod - pointer to top left of matrix storing reflection vectors (comes from
 *  the bidiag_seq routine)
 *
 * Output:
```

```
 *      V - the right orthogonal matrix in the bidiagonal decomposition
 *
 *                              A = U B V^T
 *
 * where B is bidiagonal
 *
 ************************************************************/

        int mn;
        int j_start;
        int num_refs;
        double inner_prod;
        if (m<n){
                mn = m;
                num_refs = mn;
        }
        else{
                mn = n;
                num_refs = mn-1;
        }
        // set the ith column of V to Ve_i
        for (int i=0; i < n; i++){
                set_vec_to_zero(n,V+i*n);
                V[i+i*n] = 1;
                if (i < num_refs){
                        j_start = i-1;
                }
                else{
                        j_start = num_refs-1;
                }
                for (int j=j_start; j >= 0; j--){
                        inner_prod = dot_prod_mat_row_with_vec(m,n,n-j-1,A_mod+j+(j+1)*m,V
                            +j+1+i*n);
                        for (int k=j+1; k<n; k++){
                                V[k+i*n] -= 2*A_mod[j+k*m]*inner_prod;
                        }
                }
        }
        return;
}
```

bidiagCode/bidiag_par.c

```
/*************************************************************************
 * Copyright (C) 2012 Travis Askham, Steven Delong
 * Permission is hereby granted, free of charge, to any person obtaining
 * a copy of this software and associated documentation files (the
 * "Software"), to deal in the Software without restriction, including
 * without limitation the rights to use, copy, modify, merge, publish,
 * distribute, sublicense, and/or sell copies of the Software, and to
 * permit persons to whom the Software is furnished to do so, subject to
 * the following conditions:
 *
 * The above copyright notice and this permission notice shall be
 * included in all copies or substantial portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
 * EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
 * MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
 * NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS
 * BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN
 * ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN
 * CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
 * SOFTWARE.
```

```
*****************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "matrix_helper.h"
#include "cl-helper.h"
#include "bidiag_par.h"

void bidiag_par(int m, int n, double*restrict A, double*restrict alpha, double*restrict
    beta){

/*******************************************************
 * Function: bidiag_par
 *
 * Author: Travis Askham (12/20/2012)
 *
 * Description: reduces the input matrix to bidiagonal form
 * using a parallel algorithm. The reduction is accomplished via
 * orthogonal transformations. The compute device is chosen interactively
 * or you can hardcode it below in the "OPENCL CONTEXT, QUEUE" section.
 * The create_context_on function accomplishes this. A few examples
 * are commented out in that section to see the usage. Also, uncommenting
 * "print_platforms_devices" will allow you to see what's available on
 * your machine. The function requires that double precision floating
 * point is available on the selected compute device.
 *
 * Input
 *
 *      m - the number of rows in the matrix A
 *      n - the number of columns in the matrix A
 *      A - the matrix, given in column major order
 *
 * Output
 *
 *      alpha - the diagonal of the resulting bidiagonal matrix
 *                  its length is min(m,n)
 *      beta - the superdiagonal of the resulting bidiagonal matrix
 *                  its length is min(m,n)-1 if n <= m or min(m,n) if n > m
 *      A - has been overwritten with the information necessary to
 *              reconstruct the orthogonal transformations. In particular
 *              the i-th left Householder reflector is stored in the partial
 *              column starting at A[i-1,i-1]. Likewise, the i-th right
 *      Householder reflector is stored in the partial column starting
 *      at A[i-1,i] (this is somewhat standard).
 *
 * Parameters
 *
 *      The work group sizes for the various types of CL kernels can be set
 *  at the top of the file. These can be tuned for the particular system
 *  running the code.
 *
 *      wgdim_red - the number of work items for the reduction kernels which
 *                          calculate the norm of the current column or row.
 *  work_per_item_red - how much work is done by each work item in the
 *                                  above kernel before reduction starts
 *  wgdim_upd - the number of work items for the kernels which scale and
 *                  otherwise modify the current column or row to form the
 *                  current reflection vector
 *      work_per_item_upd - how many entries each work item updates in the
 *                                  above kernels
 *  ldim_dp - the dimensions of the 2d work groups which calculate the
 *                      inner products of the reflection vector with the columns
 *                      (left Householder) or rows (right Householder) of the
```

```
*                       current working set.
*  ldim_update_mat - the dimensions of the 2d work groups which update
*                                       the working set after the inner products have
    been
*                                       calculated.
*
* Necessary OpenCL Files
*
*  cl-helper.c - library of helper functions to simplify OpenCL (written
*                                  by Andreas Kloeckner)
*  cl-helper.h - header file for cl-helper.c
*
* ****************************************************/
      int max_mn = m;
      int mn = n;
      int len_beta;
      if (m < n){
              max_mn = n;
              mn = m;
              len_beta = mn;
      }
      else{
              len_beta = mn-1;
      }

      /**************************************
       *
       * OPENCL WORK GROUP SIZE PARAMETERS
       *
       * ************************************/

       // Single column/row reduction kernels
       int wgdim_red = 8;
       int work_per_item_red = 4;

       // Single column/row update kernels
       int wgdim_upd = 4;
       int work_per_item_upd = 4;

       // Tiny no reflect kernel (for edge cases)
       int wgdim_nr = 1;
       int work_per_item_nr = 1;

       // Right and left dot product kernels
       size_t ldim_dp[] = { 8, 8 };

       // Submatrix update kernels
       size_t ldim_update_mat[] = { 16, 16 };

      /**************************************
       *
       * OPENCL CONTEXT, QUEUE
       *
       * ************************************/

      //print_platforms_devices();

      cl_context ctx;
      cl_command_queue queue;

      //create_context_on("Advanced Micro Devices, Inc.", "Turks", 0, &ctx, &queue, 0);
      //create_context_on("Intel", NULL, 0, &ctx, &queue, 0);
      create_context_on("INTERACTIVE", NULL, 0, &ctx, &queue, 0);
      //create_context_on("NVIDIA Corporation", NULL, 0, &ctx, &queue, 0);
```

```
    // pointer to the text of the kernel file

    char *knl_text;

    /*****************************
     *
     * KERNEL COMPILATION OPTIONS
     *
     * ***************************/

    // options string for kernel from string, tells the kernel its local dimensions

// Single column/row reduction kernels
    const char base[] = "-DLOC_SIZE=";
    char options_str_reduction [ 20 ];
    int size = work_per_item_red*wgdim_red;
    sprintf(options_str_reduction, "%s%d", base, size);

    // Update a column or row
    char options_str_update [ 20 ];
    size = wgdim_upd; //only need to store a copy of the innerprod
    sprintf(options_str_update, "%s%d", base, size);

    // Dot product kernels
    const char base0[] = "-DLOC_SIZE0=";
    const char base1[] = "-DLOC_SIZE1=";
    char options_str_dp [ 40 ];
    size = ldim_dp[0];
    sprintf(options_str_dp, "%s%d ", base0, size);
    size = ldim_dp[1];
    sprintf(options_str_dp, "%s%s%d ", options_str_dp,base1, size);

    // Matrix update kernels
    char options_str_update_mat [ 40 ];
    size = ldim_update_mat[0];
    sprintf(options_str_update_mat, "%s%d ", base0, size);
    size = ldim_update_mat[1];
    sprintf(options_str_update_mat, "%s%s%d ", options_str_update_mat,base1, size);

    /*************************************
     *
     * COMPILE OPENCL KERNELS
     *
     * ***********************************/

    // norm square mat col kernel [reduction]
    knl_text = read_file("normsq_matcol.cl");
    cl_kernel knl_normsq_matcol = kernel_from_string(ctx, knl_text,
            "normsq_matcol", options_str_reduction);
    free(knl_text);

    // norm square mat row kernel [reduction]
    knl_text = read_file("normsq_matrow.cl");
    cl_kernel knl_normsq_matrow = kernel_from_string(ctx, knl_text,
            "normsq_matrow", options_str_reduction);
    free(knl_text);

    // sum kernel [reduction]
    knl_text = read_file("sum.cl");
    cl_kernel knl_sum = kernel_from_string(ctx, knl_text,
            "sum", options_str_reduction);
    free(knl_text);

    // update and scale mat col [update]
```

```
knl_text = read_file("update_scale_matcol.cl");
cl_kernel knl_update_scale_matcol = kernel_from_string(ctx, knl_text,
        "update_scale_matcol", options_str_update);
free(knl_text);

// update and scale mat row [update]
knl_text = read_file("update_scale_matrow.cl");
cl_kernel knl_update_scale_matrow = kernel_from_string(ctx, knl_text,
        "update_scale_matrow", options_str_update);
free(knl_text);

// no reflect [single workitem/group]
knl_text = read_file("sanders.cl");
cl_kernel knl_sanders = kernel_from_string(ctx, knl_text,
        "sanders", NULL);
free(knl_text);

// left_dotprods [dot_prods]
knl_text = read_file("left_dotprods.cl");
cl_kernel knl_left_dotprods = kernel_from_string(ctx, knl_text,
        "left_dotprods", options_str_dp);
free(knl_text);

// right_dotprods [dot_prods]
knl_text = read_file("right_dotprods.cl");
cl_kernel knl_right_dotprods = kernel_from_string(ctx, knl_text,
        "right_dotprods", options_str_dp);
free(knl_text);

// left update mat [update_mat]
knl_text = read_file("left_update_mat.cl");
cl_kernel knl_left_update_mat = kernel_from_string(ctx, knl_text,
        "left_update_mat", options_str_update_mat);
free(knl_text);

// right update mat [update_mat]
knl_text = read_file("right_update_mat.cl");
cl_kernel knl_right_update_mat = kernel_from_string(ctx, knl_text,
        "right_update_mat", options_str_update_mat);
free(knl_text);

/*************************************
 *
 * ALLOCATE MEMORY ON THE DEVICE
 *
 * ************************************/
cl_int status;

// to store the matrix A in memory
cl_mem buf_A = clCreateBuffer(ctx, CL_MEM_READ_WRITE,
  sizeof(double) *m*n, 0, &status);
CHECK_CL_ERROR(status, "clCreateBuffer for Matrix A");

// for alpha and beta on device
cl_mem buf_alpha = clCreateBuffer(ctx, CL_MEM_READ_WRITE,
  sizeof(double) *mn, 0, &status);
CHECK_CL_ERROR(status, "clCreateBuffer for alpha");
cl_mem buf_beta = clCreateBuffer(ctx, CL_MEM_READ_WRITE,
  sizeof(double) *len_beta, 0, &status);
CHECK_CL_ERROR(status, "clCreateBuffer for beta");

// for work/storage
cl_mem buf_scratch = clCreateBuffer(ctx, CL_MEM_READ_WRITE,
  sizeof(double) * max_mn, 0, &status);
```

```
CHECK_CL_ERROR(status, "clCreateBuffer for scratch work 1");
cl_mem buf_scratch2 = clCreateBuffer(ctx, CL_MEM_READ_WRITE,
  sizeof(double) * max_mn, 0, &status);
CHECK_CL_ERROR(status, "clCreateBuffer for scratch work 2");
cl_mem buf_temp = clCreateBuffer(ctx, CL_MEM_READ_WRITE,
  sizeof(double), 0, &status);
CHECK_CL_ERROR(status, "clCreateBuffer for temp");

/*********************************************
 *
 * WRITE THE MATRIX A TO ITS BUFFER
 *
 * ******************************************/

CALL_CL_GUARDED(clEnqueueWriteBuffer, (
        queue, buf_A, /*blocking*/ CL_TRUE, /*offset*/ 0,
        sizeof(double) * m*n, A,
        0, NULL, NULL));

/************************************************
 *
 * MAIN LOOP FOR APPLYING LEFT AND RIGHT HOUSEHOLDER REFLECTORS
 * TO REDUCE THE MATRIX TO BIDIAGONAL FORM
 *
 * **********************************************/

for (int i=0; i< mn-1; i++){

        // Takes the current column, calculates the appropriate reflector,
        // and stores it in place
        find_reflector_and_scale_col(m, n, mn, i,
                        work_per_item_red, wgdim_red, work_per_item_upd, wgdim_upd
                            ,
                        ctx, queue,
                        knl_normsq_matcol, knl_update_scale_matcol, knl_sum,
                        buf_A, buf_alpha,
                        buf_scratch, buf_scratch2, buf_temp);

        // Applies the reflector from the previous step to the working set
        // i.e. the submatrix whose top left is A[i,i+1]
        left_householder_par_2d(m,n,mn,i, ldim_dp, ldim_update_mat,
                        ctx, queue,
                        knl_left_dotprods, knl_left_update_mat,
                        buf_A, buf_scratch);

        if ( i < n-2){
                // Takes the current row, calculates the appropriate reflector,
                // and stores it in place
                find_reflector_and_scale_row(m, n, mn, i,
                        work_per_item_red, wgdim_red, work_per_item_upd, wgdim_upd
                            ,
                        ctx, queue,
                        knl_normsq_matrow, knl_update_scale_matrow,
                        knl_sum,
                        buf_A, buf_beta,
                        buf_scratch, buf_scratch2, buf_temp);
                // Applies the reflector from the previous step to the working set
                // i.e. the submatrix whose top left is A[i+1,i+1]
                right_householder_par_2d(m,n,mn,i, ldim_dp, ldim_update_mat,
                        ctx, queue,
                        knl_right_dotprods, knl_right_update_mat,
                        buf_A, buf_scratch);
        }
        else{
```

```
                // EDGE CASE: there is no more to reduce on the right

                int mat_loc = i+(i+1)*m;
                int vec_loc = i;

                // no reflection on right
                no_reflect(m,n,mat_loc,vec_loc,work_per_item_nr,wgdim_nr,
                                        ctx,queue,knl_sanders,
                                        buf_A,buf_beta);
        }
}

if ( n >= m+1 ){

        // EDGE CASE

        // do right householder mod to this row of A
        // no need to apply to the rest of the rows (there aren't any more)
        int temp_offset = mn-1;
        find_reflector_and_scale_row(m, n, mn, temp_offset,
                work_per_item_red, wgdim_red, work_per_item_upd, wgdim_upd,
                ctx, queue,
                knl_normsq_matrow, knl_update_scale_matrow,
                knl_sum,
                buf_A, buf_beta,
                buf_scratch, buf_scratch2, buf_temp);

        int mat_loc = mn-1+(mn-1)*m;
        int vec_loc = mn-1;

        // no reflection on left (it's already lower triangular)
        no_reflect(m,n,mat_loc,vec_loc,work_per_item_nr,wgdim_nr,
                                ctx,queue,knl_sanders,
                                buf_A,buf_alpha);
}
else {

        // EDGE CASE

        // do the left Householder reflection (no need to apply it to
        // the rest of the columns .. as there aren't any more)
        int temp_offset = mn-1;
        find_reflector_and_scale_col(m, n, mn, temp_offset,
                        work_per_item_red, wgdim_red, work_per_item_upd, wgdim_upd
                            ,
                        ctx, queue,
                        knl_normsq_matcol, knl_update_scale_matcol, knl_sum,
                        buf_A, buf_alpha,
                        buf_scratch, buf_scratch2, buf_temp);
}

/***********************************************
 *
 * GET THE MODIFIED A AND THE DIAGONALS OF B BACK FROM DEVICE
 *
 * ********************************************/

CALL_CL_GUARDED(clEnqueueReadBuffer, (
                queue, buf_alpha, /*blocking*/ CL_TRUE, /*offset*/ 0,
                mn* sizeof(double), alpha,
                0, NULL, NULL));

CALL_CL_GUARDED(clEnqueueReadBuffer, (
                queue, buf_beta, /*blocking*/ CL_TRUE, /*offset*/ 0,
```

```
                        len_beta* sizeof(double), beta,
                        0, NULL, NULL));

        CALL_CL_GUARDED(clEnqueueReadBuffer, (
                        queue, buf_A, /*blocking*/ CL_TRUE, /*offset*/ 0,
                        m*n* sizeof(double), A,
                        0, NULL, NULL));

        CALL_CL_GUARDED(clFinish, (queue));

        /*********************************************
         *
         * MEMORY CLEAN UP FOR OPENCL OBJECTS
         * AND DEVICE BUFFERS
         *
         * ********************************************/

        CALL_CL_GUARDED(clReleaseMemObject, (buf_A));
        CALL_CL_GUARDED(clReleaseMemObject, (buf_scratch));
        CALL_CL_GUARDED(clReleaseMemObject, (buf_scratch2));
        CALL_CL_GUARDED(clReleaseMemObject, (buf_temp));
        CALL_CL_GUARDED(clReleaseMemObject, (buf_alpha));
        CALL_CL_GUARDED(clReleaseMemObject, (buf_beta));
        CALL_CL_GUARDED(clReleaseKernel, (knl_normsq_matcol));
        CALL_CL_GUARDED(clReleaseKernel, (knl_update_scale_matcol));
        CALL_CL_GUARDED(clReleaseKernel, (knl_update_scale_matrow));
        CALL_CL_GUARDED(clReleaseKernel, (knl_normsq_matrow));
        CALL_CL_GUARDED(clReleaseKernel, (knl_sum));
        CALL_CL_GUARDED(clReleaseKernel, (knl_sanders));
        CALL_CL_GUARDED(clReleaseKernel, (knl_left_update_mat));
        CALL_CL_GUARDED(clReleaseKernel, (knl_left_dotprods));
        CALL_CL_GUARDED(clReleaseKernel, (knl_right_update_mat));
        CALL_CL_GUARDED(clReleaseKernel, (knl_right_dotprods));
        CALL_CL_GUARDED(clReleaseCommandQueue, (queue));
        CALL_CL_GUARDED(clReleaseContext, (ctx));
        return;
}

void no_reflect( int m, int n, int mat_loc, int vec_loc,
                                int work_per_item_nr, int wgdim_nr,
                                cl_context ctx, cl_command_queue queue,
                                cl_kernel knl_sanders,
                                cl_mem buf_mat, cl_mem buf_vec){

/*********************************************************
 * Function: no_reflect
 *
 * Author: Travis Askham (12/20/2012)
 *
 * Description: This function is used for edge cases in which there is
 * no reflection on either the left or right for the current step. In
 * this case, 0 is stored for the reflection vector and the value of
 * A is stored in the corresponding location in B (the bidiagonal
 * matrix). The result is
 *
 *      alpha[vec_loc] or beta[vec_loc] = A[mat_loc]
 *      A[mat_loc] --> 0
 *
 * where buf_vec deterimines if it is alpha or beta.
 *
 * Input:
 *      m - number of rows in matrix
 *      n - number of columns in matrix
 *  mat_loc - as above
```

```
 *       vec_loc - as above
 *       buf_A - the on-chip matrix A
 *   buf_vec - as above
 *   OpenCL Objects: should always be called with the kernels, context,
 *   and queue in the function prototype. The integer wgdim_nr sets the
 *   size of the work group dimensions and work_per_item determines how
 *   much work is done by each work_item (should always be 1).
 *
 ************************************************************/

        size_t ldim[] = { wgdim_nr };
        size_t gdim[] = { wgdim_nr };

        SET_6_KERNEL_ARGS( knl_sanders, buf_mat, buf_vec, m, n, mat_loc, vec_loc);

        CALL_CL_GUARDED(clEnqueueNDRangeKernel,
                        (queue, knl_sanders,
                         /*dimensions*/1, NULL, gdim, ldim,
                         0, NULL, NULL));

        return;
}

void left_householder_par_2d(int m, int n, int mn, int offset,
                             size_t ldim_dp[], size_t ldim_update_mat[],
                             cl_context ctx, cl_command_queue queue,
                             cl_kernel knl_left_dotprods, cl_kernel knl_left_update_mat
                                 ,
                             cl_mem buf_A, cl_mem buf_scratch){

/************************************************************
 * Function: left_householder_par_2d
 *
 * Author: Travis Askham (12/20/2012)
 *
 * Description: Let v be the reflector to be applied on the left. The
 * result of this function is
 *
 *       A --> (I - 2v v^T) A
 *
 * the fact that v is zero in its first several entries accounts for the
 * offsets mentioned below.
 *
 * Input:
 *       m - number of rows in matrix
 *       n - number of columns in matrix
 *   mn - the minimum of m and n
 *       offset - the reflector is stored in the partial column starting at
 *                       A[offset,offset]
 *       buf_A - the on-chip matrix A
 *   buf_scratch - on-chip memory for temporary storage of the inner
 *                         products of the reflector with the columns of the submatrix
 *                                 whose top-left is A[offset,offset+1]
 *   OpenCL Objects: should always be called with the kernels, context,
 *   and queue in the function prototype. The length 2 arrays ldim_dp,
 *   ldim_update_mat give the local work group dimensions for the
 *   corresponding OpenCL kernels.
 *
 *
 ************************************************************/

        // KERNEL LAUNCH INFO FOR DOT PRODUCTS
        size_t total_size1 = ((n-offset-1 + ldim_dp[1] - 1)
                /(ldim_dp[1]))*ldim_dp[1];
```

```
        size_t gdim[] = { ldim_dp[0]*1, total_size1 };

        // CALCULATES THE DOT PRODUCTS AND STORES THEM IN buf_scratch
        SET_6_KERNEL_ARGS(knl_left_dotprods, buf_A, buf_scratch,
                m,n,offset,offset);
        CALL_CL_GUARDED(clEnqueueNDRangeKernel,
                        (queue, knl_left_dotprods,
                         /*dimensions*/2, NULL, gdim, ldim_dp,
                         0, NULL, NULL));

        // KERNEL LAUNCH INFO FOR MATRIX UPDATES
        size_t total_size0 = ((m-offset+ ldim_update_mat[0]-1)/(ldim_update_mat[0])) *
            ldim_update_mat[0];
        total_size1 = ((n-offset-1 + ldim_update_mat[1] - 1)/(ldim_update_mat[1])) *
            ldim_update_mat[1];

        gdim[0] = total_size0;
        gdim[1] = total_size1;

        // GIVEN THE INNERPRODUCT, APPLIES REFLECTOR TO EACH COLUMN OF THE
        // SUBMATRIX STARTING AT A[offset,offset+1]
        SET_6_KERNEL_ARGS(knl_left_update_mat, buf_A, buf_scratch,
                          m,n,offset,offset);
        CALL_CL_GUARDED(clEnqueueNDRangeKernel,
                (queue, knl_left_update_mat,
                 /*dimensions*/2, NULL, gdim, ldim_update_mat,
                 0, NULL, NULL));
        return;
}

void right_householder_par_2d(int m, int n, int mn, int offset,
                              size_t ldim_dp[], size_t ldim_update_mat[],
                              cl_context ctx, cl_command_queue queue,
                              cl_kernel knl_right_dotprods, cl_kernel
                                  knl_right_update_mat,
                              cl_mem buf_A, cl_mem buf_scratch){

/*************************************************************
 * Function: right_householder_par_2d
 *
 * Author: Travis Askham (12/20/2012)
 *
 * Description: Let v be the reflector to be applied on the right. The
 * result of this function is
 *
 *      A --> ((I - 2v v^T) A^T)^T
 *
 * the fact that v is zero in its first several entries accounts for the
 * offsets mentioned below.
 *
 * Input:
 *      m - number of rows in matrix
 *      n - number of columns in matrix
 *  mn - the minimum of m and n
 *      offset - the reflector is stored in the partial row starting at
 *                    A[offset,offset+1]
 *      buf_A - the on-chip matrix A
 *  buf_scratch - on-chip memory for temporary storage of the inner
 *                        products of the reflector with the rows of the submatrix
 *                                  whose top-left is A[offset+1,offset+1]
 *  OpenCL Objects: should always be called with the kernels, context,
 *  and queue in the function prototype. The length 2 arrays ldim_dp,
 *  ldim_update_mat give the local work group dimensions for the
 *  corresponding OpenCL kernels.
```

```
 *
 *
 ******************************************************/

        // KERNEL LAUNCH INFO FOR DOT PRODUCTS
        size_t total_size0 = ((m-offset-1 + ldim_dp[0] - 1)
                /(ldim_dp[0]))*ldim_dp[0];
        size_t gdim[] = { total_size0, ldim_dp[1]*1 };

        int col_offset = offset+1;
        // CALCULATES THE DOT PRODUCTS AND STORES THEM IN buf_scratch
        SET_6_KERNEL_ARGS(knl_right_dotprods, buf_A, buf_scratch,
                m,n,offset,col_offset);
        CALL_CL_GUARDED(clEnqueueNDRangeKernel,
                        (queue, knl_right_dotprods,
                         /*dimensions*/2, NULL, gdim, ldim_dp,
                         0, NULL, NULL));

        // KERNEL LAUNCH INFO FOR MATRIX UPDATES
        total_size0 = ((m-offset-1+ ldim_update_mat[0]-1)/(ldim_update_mat[0])) *
            ldim_update_mat[0];
        size_t total_size1 = ((n-col_offset + ldim_update_mat[1] - 1)/(ldim_update_mat[1])
            ) * ldim_update_mat[1];

        gdim[0] = total_size0;
        gdim[1] = total_size1;

        // GIVEN THE INNERPRODUCT, APPLIES REFLECTOR TO EACH ROW OF THE
        // SUBMATRIX STARTING AT A[offset+1,offset+1]
        SET_6_KERNEL_ARGS(knl_right_update_mat, buf_A, buf_scratch,
                          m,n,offset,col_offset);
        CALL_CL_GUARDED(clEnqueueNDRangeKernel,
                (queue, knl_right_update_mat,
                 /*dimensions*/2, NULL, gdim, ldim_update_mat,
                 0, NULL, NULL));
        return;
}

void find_reflector_and_scale_col(int m, int n, int mn, int col,
                                  int work_per_item_red, int wgdim_red, int
                                      work_per_item_upd, int wgdim_upd,
                                  cl_context ctx, cl_command_queue queue,
                                  cl_kernel knl_normsq_matcol,cl_kernel
                                      knl_update_scale_matcol,
                                  cl_kernel knl_sum,
                                  cl_mem buf_A, cl_mem buf_alpha,
                                  cl_mem buf_scratch, cl_mem buf_scratch2, cl_mem buf_temp){

/******************************************************
 * Function: find_reflector_and_scale_col
 *
 * Author: Travis Askham (12/20/2012)
 *
 * Description: This function creates the left Householder reflector for
 * the current step and stores it in place. Let v be the partial column
 * starting at A[col,col]. The result of this function is
 *
 *              alpha[col] --> - sign(v[0]) || v ||
 *      v --> normalize(sign(v[0]) || v || e_0 + v)
 *
 * where || v || indicates the 2-norm of the partial column v. The vector
 * e_0 is the standard basis vector with 1 in its first entry. The norm
 * of the partial column is calculated via a reduction.
 *
```

```
* Input:
*      m - number of rows in matrix
*      n - number of columns in matrix
*  col - as above
*      buf_A - the on-chip matrix A
*  buf_alpha - on-chip storage of the diagonal of B
*  buf_scratch(2) - on-chip scratch memory for accomplishing the reductions
*  buf_temp - on-chip memory for storing A[col,col]
*  OpenCL Objects: should always be called with the kernels, context,
*  and queue in the function prototype. The integers  work_per_item_red,
*  wgdim_red, work_per_item_upd, wgdim_upd determine the work group
*  dimensions and the amount of work done per work item for the
*  corresponding kernels.
*
*
**************************************************/

        int offset = col;
        size_t ldim[] = { wgdim_red };
        size_t total_size = ((m-offset + ldim[0]*work_per_item_red - 1)
                /(ldim[0]*work_per_item_red))*ldim[0];
        int current_num_groups = ((m-offset + ldim[0]*work_per_item_red - 1)
                /(ldim[0]*work_per_item_red));
        size_t gdim[] = { total_size };

        cl_mem in = buf_A;
        cl_mem out = buf_scratch;

        // INITIAL REDUCTION FOR NORM SQUARED CALCULATION
        // This pass squares the entries and adds them
        // It also outputs the first entry of the partial column of A to buf_temp
        SET_8_KERNEL_ARGS(knl_normsq_matcol, in, out, buf_temp,
                m,n,col,col,work_per_item_red);
        CALL_CL_GUARDED(clEnqueueNDRangeKernel,
                (queue, knl_normsq_matcol,
                 /*dimensions*/1, NULL, gdim, ldim,
                 0, NULL, NULL));
        cl_mem swap;
        in = buf_scratch;
        out = buf_scratch2;
        int odd = 1;
        int write_offset = 0;
        // Further reduction steps which simply add the numbers together
        // not necessary to square again
        while (current_num_groups > 1){
                SET_5_KERNEL_ARGS(knl_sum, in, out,current_num_groups,
                     work_per_item_red,write_offset);
                // FIXME set to 1
                current_num_groups = ((current_num_groups +
                                  ldim[0]*work_per_item_red - 1)
                                 /(ldim[0]*work_per_item_red));
                gdim[0] = current_num_groups*ldim[0];
                CALL_CL_GUARDED(clEnqueueNDRangeKernel,
                         (queue, knl_sum,
                          /*dimensions*/1, NULL, gdim, ldim,
                          0, NULL, NULL));
                odd++;
                swap = out;
                out = in;
                in = swap;
        } // norm is now stored in buff_scratch
          // if odd is odd, buff_scratch2 if odd is even
        if ( odd%2 == 0)
                in = buf_scratch2;
```

```
        else
                in = buf_scratch;
        ldim[0] = wgdim_upd;
        total_size = ((m-offset + ldim[0]*work_per_item_upd - 1)
                /(ldim[0]*work_per_item_upd))*ldim[0];
        gdim[0] = total_size;
        // UDPATE THE COLUMN OF A
        SET_9_KERNEL_ARGS(knl_update_scale_matcol, buf_A, buf_alpha, in, buf_temp,
                m,n,col,col,work_per_item_upd);
        CALL_CL_GUARDED(clEnqueueNDRangeKernel,
                (queue, knl_update_scale_matcol,
                 /*dimensions*/1, NULL, gdim, ldim,
                 0, NULL, NULL));
        return;
}


void find_reflector_and_scale_row(int m, int n, int mn, int row,
                                  int work_per_item_red,
                                  int wgdim_red, int work_per_item_upd, int wgdim_upd,
                                  cl_context ctx, cl_command_queue queue,
                                  cl_kernel knl_normsq_matrow,cl_kernel
                                      knl_update_scale_matrow,
                                  cl_kernel knl_sum,
                                  cl_mem buf_A, cl_mem buf_beta,
                                  cl_mem buf_scratch, cl_mem buf_scratch2, cl_mem buf_temp){

/*********************************************************
 * Function: find_reflector_and_scale_row
 *
 * Author: Travis Askham (12/20/2012)
 *
 * Description: This function creates the right Householder reflector for
 * the current step and stores it in place. Let r be the partial row
 * starting at A[row,row+1]. The result of this function is
 *
 *              beta[row] -->   - sign(r[0]) || r ||
 *      r --> normalize(sign(r[0]) || r || e_0 + r)
 *
 * where || r || indicates the 2-norm of the partial row r. The vector
 * e_0 is the standard basis vector with 1 in its first entry. The norm
 * of the partial row is calculated via a reduction.
 *
 * Input:
 *      m - number of rows in matrix
 *      n - number of columns in matrix
 *  row - as above
 *      buf_A - the on-chip matrix A
 *  buf_beta - on-chip storage of the superdiagonal of B.
 *  buf_scratch(2) - on-chip scratch memory for accomplishing the reductions
 *  buf_temp - on-chip memory for storing A[col,col]
 *  OpenCL Objects: should always be called with the kernels, context,
 *  and queue in the function prototype. The integers  work_per_item_red,
 *  wgdim_red, work_per_item_upd, wgdim_upd determine the work group
 *  dimensions and the amount of work done per work item for the
 *  corresponding kernels.
 *
 *
 *********************************************************/

        int offset = row+1;
        size_t ldim[] = { wgdim_red };
        size_t total_size = ((n-offset + ldim[0]*work_per_item_red - 1)
                /(ldim[0]*work_per_item_red))*ldim[0];
```

```
int current_num_groups = ((n-offset + ldim[0]*work_per_item_red - 1)
        /(ldim[0]*work_per_item_red));
size_t gdim[] = { total_size };

cl_mem in = buf_A;
cl_mem out = buf_scratch;

// INITIAL REDUCTION FOR NORM SQUARED CALCULATION
// This pass squares the entries and adds them
// It also outputs the first entry of the partial row of A to buf_temp
SET_8_KERNEL_ARGS(knl_normsq_matrow, in, out, buf_temp,
        m,n,row,offset,work_per_item_red);
CALL_CL_GUARDED(clEnqueueNDRangeKernel,
        (queue, knl_normsq_matrow,
         /*dimensions*/1, NULL, gdim, ldim,
         0, NULL, NULL));

double norm;



cl_mem swap;
in = buf_scratch;
out = buf_scratch2;

int odd = 1;
int write_offset = 0;

// Further reduction steps which simply add the numbers together
// not necessary to square again
while (current_num_groups > 1){
        SET_5_KERNEL_ARGS(knl_sum, in, out,current_num_groups,
                          work_per_item_red,write_offset);
        // FIXME set to 1
        current_num_groups = ((current_num_groups +
                               ldim[0]*work_per_item_red - 1)
                               /(ldim[0]*work_per_item_red));
        gdim[0] = current_num_groups*ldim[0];
        CALL_CL_GUARDED(clEnqueueNDRangeKernel,
                (queue, knl_sum,
                 /*dimensions*/1, NULL, gdim, ldim,
                 0, NULL, NULL));

        odd++;
        swap = out;
        out = in;
        in = swap;
} // norm is now stored in buff_scratch
  // if odd is odd, buff_scratch2 if odd is even
if ( odd%2 == 0)
        in = buf_scratch2;
else
        in = buf_scratch;
ldim[0] = wgdim_upd;
total_size = ((n-offset + ldim[0]*work_per_item_upd - 1)
        /(ldim[0]*work_per_item_upd))*ldim[0];

gdim[0] = total_size;

// UDPATE THE ROW OF A
SET_9_KERNEL_ARGS(knl_update_scale_matrow, buf_A, buf_beta, in, buf_temp,
        m,n,row,offset,work_per_item_upd);
CALL_CL_GUARDED(clEnqueueNDRangeKernel,
        (queue, knl_update_scale_matrow,
```

```
                /*dimensions*/1, NULL, gdim, ldim,
                0, NULL, NULL));

        return;

}

void form_u_par(int m, int n, const double*restrict A_mod, double*restrict U){

/************************************************************
 * Function: form_u_par
 *
 * Author: Travis Askham (12/20/2012)
 *
 * Input:
 *      m - number of rows in matrix
 *      n - number of columns in matrix
 *      A_mod - pointer to top left of matrix storing reflection vectors (comes from
 *  the bidiag_seq routine)
 *
 * Output:
 *      U - the left orthogonal matrix in the bidiagonal decomposition
 *
 *                              A = U B V^T
 *
 * where B is bidiagonal
 *
 * NOTE: not a parallel routine. Exactly the same as form_u found in bidiag.c
 *
 ************************************************************/

        int mn;
        int j_start;
        double inner_prod;
        if (m<n){
                mn = m;
        }
        else{
                mn = n;
        }

        // set the ith column of U to Ue_i
        for (int i=0; i < m; i++){
                set_vec_to_zero(m,U+i*m);
                U[i+i*m] = 1;
                if (i < mn-1 ){
                        j_start = i;
                }
                else{
                        j_start = mn-1;
                }
                for (int j=j_start; j >= 0; j--){
                        inner_prod = dot_prod(m-j,U+j+i*m,A_mod+j+j*m);
                        for (int k=j; k<m; k++){
                                U[k+i*m] -= 2*A_mod[k+j*m]*inner_prod;
                        }
                }
        }
        return;
}

void form_v_par(int m, int n, const double*restrict A_mod, double*restrict V){

/************************************************************
```

```
 * Function: form_v_par
 *
 * Author: Travis Askham (12/20/2012)
 *
 * Input:
 *      m - number of rows in matrix
 *      n - number of columns in matrix
 *      A_mod - pointer to top left of matrix storing reflection vectors (comes from
 *  the bidiag_seq routine)
 *
 * Output:
 *      V - the right orthogonal matrix in the bidiagonal decomposition
 *
 *                                  A = U B V^T
 *
 * where B is bidiagonal
 *
 * NOTE: Not a parallel routine. Exactly the same as form_v found in bidiag.c
 *
 ************************************************************/

        int mn;
        int j_start;
        int num_refs;
        double inner_prod;
        if (m<n){
                mn = m;
                num_refs = mn;
        }
        else{
                mn = n;
                num_refs = mn-1;
        }

        // set the ith column of V to Ve_i
        for (int i=0; i < n; i++){
                set_vec_to_zero(n,V+i*n);
                V[i+i*n] = 1;
                if (i < num_refs){
                        j_start = i-1;
                }
                else{
                        j_start = num_refs-1;
                }
                for (int j=j_start; j >= 0; j--){
                        inner_prod = dot_prod_mat_row_with_vec(m,n,n-j-1,
                                                A_mod+j+(j+1)*m,V+j+1+i*n);
                        for (int k=j+1; k<n; k++){
                                V[k+i*n] -= 2*A_mod[j+k*m]*inner_prod;
                        }
                }
        }
        return;
}

void multV(int m,int n, int vecnum, double*restrict A_mod, double*restrict X,double*
    restrict V){
    /****************************************
     * Author: Steven Delong
     *
     * Description: creates a right singular vector V from the reflectors stored in A
         after
     * bidiagonalization and from the singular vectors found in X after running
     * CalcRightSingularVectors
```

```
     *
     * inputs:
     *       m - row dimension of the original matrix A, output will be m x m
     *       mn - minimum dimension, min(m,n)
     *       vecnum - number of the vector to compute.
     *       A_mod - stores reflectors, it will be the TRANSPOSE of the overwritten A
         created
     *           by the bidiagonalization function (e.g. bidiag_seq)
     *      X - right singular vectors for the reduced bidiagonal matrix, output from
     *           CalcRightSingularVectors
     *
     * Outputs:
     *        V - the vecnum'th right singular vector of A
     ************************************************/

    int mn, num_refs,rest;
    double inner_prod;
    if (m<n){
        mn = m;
        num_refs = mn;
    }
    else{
        mn = n;
        num_refs = mn-1;
    }

    // first set V to vecnumth vector of X, padded with zeros
    for(int i = 0; i < num_refs + 1; ++i){
        V[i] = X[vecnum*mn + i];
    }
    if(num_refs + 1 < n){
        for(int i = num_refs + 1; i < n; ++i){
            V[i] = 0.0;
        }
    }

    for (int j=num_refs + 1; j >= 0; j--){
        inner_prod = dot_prod(n-j-1,A_mod+j+1+j*m,V+j+1);
        for (int k=j+1; k<n; k++){
            V[k] -= 2*A_mod[j*m+ k]*inner_prod;
        }
    }
}

void multU(int m,int n, int vecnum, double*restrict A_mod, double*restrict Y,double*
    restrict U){
    /****************************************
     * Author: Steven Delong
     *
     * Description: creates a left singular vector U from the reflectors stored in A after
     * bidiagonalization and from the singular vectors found in Y after running
     * RighttoLeftSingularVecrors.
     *
     * inputs:
     *       m - row dimension of the original matrix A, output will be m x m
     *       mn - minimum dimension, min(m,n)
     *       vecnum - number of the vector to compute.
     *       A_mod - stores reflectors, it will be the overwritten A created
     *           by the bidiagonalization function (e.g. bidiag_seq)
     *       Y - left singular vectors for the reduced bidiagonal matrix, output from
     *           RighttoLeftsingularvectors
     *
     * Outputs:
     *        U - the vecnum'th left singular vector of A
```

```
     ************************************************/

    int mn, num_refs;
    if (m<n){
        mn = m;
        num_refs = mn;
    }
    else{
        mn = n;
        num_refs = mn-1;
    }

    double inner_prod;
    // first set U to vecnumth vector of Y, padded with zeros
    for(int i = 0; i < mn; ++i){
        U[i] = Y[vecnum*mn + i];
    }
    if(mn  < m){
        for(int i = mn; i < m; ++i){ // shouldn't need to do this.
            U[i] = 0.0;
        }
    }

    // apply reflectors to U until we have the singular vector we want
    for (int j=num_refs; j >= 0; j--){
        inner_prod = dot_prod(m-j,U+j,A_mod+j+j*m);
        for (int k=j; k<m; k++){
            U[k] -= 2*A_mod[k+j*m]*inner_prod;
        }
    }
}
```

Appendix C

Test Machines Hardware Specifications

```
APU machine:
Number of OpenCL platforms: 1
Platform: AMD Accelerated Parallel Processing
Vendor: Advanced Micro Devices, Inc.
Version: OpenCL 1.2 AMD-APP (1268.1)
Number of devices: 2
-------------------------
Name: BeaverCreek
Version: OpenCL C 1.2
Max. Compute Units: 4
Local Memory Size: 32 KB
Global Memory Size: 256 MB
Max Alloc Size: 128 MB
Max Work-group Size: 256
Max Work-item Dims:( 256 256 256 )
-------------------------
Name: AMD A6-3670 APU with Radeon(tm) HD Graphics
Version: OpenCL C 1.2
Max. Compute Units: 4
Local Memory Size: 32 KB
Global Memory Size: 15525 MB
Max Alloc Size: 3881 MB
Max Work-group Size: 1024
-------------------------
Number of OpenCL platforms: 1
Platform: NVIDIA CUDA
Vendor: NVIDIA Corporation
Version: OpenCL 1.1 CUDA 4.2.1
Number of devices: 1
-------------------------
Name: GeForce GTX 570
Version: OpenCL C 1.1
Max. Compute Units: 15
```

```
 Local Memory Size: 48 KB
 Global Memory Size: 1279 MB
 Max Alloc Size: 319 MB
 Max Work-group Size: 1024
 Max Work-item Dims:( 1024 1024 64 )
 Max Work-item Dims:( 1024 1024 1024 )
-------------------------
```

Vita

Michael V Risley obtained a Bachelor of Science Degree in Mathematics from Virginia Commonwealth University in 2010 and the Certified Energy Manager designation from the Association of Energy Engineers in 2011. He currently works as the Assistant Energy Manager at Virginia Commonweatlh University. His duties include data analysis of energy consumption and building automation systems to help the university meets its climate action goals.